Data Restructuring as Formal Preprocessing for Machine Learning with Neural Networks

Doctoral Thesis
(Dissertation)

to be awarded the degree
Doctor rerum naturalium (Dr. rer. nat.)

submitted by
Sven Arnhold
from Haselünne, Germany

approved by the Faculty of
Mathematics/Computer Science and Mechanical Engineering,
Clausthal University of Technology,

Date of oral examination
28th of August 2015

Chairperson of the Board of Examiners
Prof. Dr. rer. nat. Jörg P. Müller,
Clausthal University of Technology

Chief Reviewer
Prof. Dr. rer. nat. habil. Sven Hartmann,
Clausthal University of Technology

Reviewer
Prof. Dr. rer. nat. habil. Barbara Hammer,
Bielefeld University

Acknowledgements

Deepest gratitude is paid to my doctoral supervisor Mr. Hartmann, who through patience and words and deeds enabled me to receive a doctorate at the Clausthal University of Technology. Without his support and constant motivation and the many helpful conversations where I got ideas and suggestions, this thesis and the underlying research would not have been possible.

Deepest gratitude is also paid to Mrs. Hammer for reviewing this thesis. She also supervised my diploma thesis at the end of my studies at the Clausthal University of Technology, and some of its results could now also be applied in this doctoral thesis.

Furthermore, I want to thank my friends, colleagues and fellow employees not only from the Department of Informatics who accompanied my path with inspiration, relief and motivation.

Finally, I would like to express special thanks to Nina, Wiebke, Arne and Hendrik for their proofreading of this thesis, Jana for her hints on its outline, the Poettering family for their kind hospitality over many years and even more the hopefully few that I unfortunately forgot to mention.

Dedicated to Sinuhe, the son of Senmut and of his wife Kipa,
and to all those who are weary of the gods and
who write for their own sake alone.[1]

[1] loosely according to Mika Waltari, "Sinuhe the Egyptian".

Zusammenfassung

Künstliche neuronale Netze werden im Bereich des maschinellen Lernens zur Nachahmung von Expertenwissen eingesetzt. Sie können als feedforward Netze zwischen Daten mit fester Struktur abbilden, als rekurrente Netze auf Daten mit sequentiellem Charakter wie z.B. Zeitreihen und als rekursive Netze zum Lernen auf Datenstrukturen wie chemischen Strukturformeln verwendet werden.

In der Praxis gestaltet sich das Training, also die Anpassung der freien Parameter, meistens schwierig. Ständiger Gegenstand der Forschung ist daher unter anderem, spezielle Netzarchitekturen zu entwickeln, die sich für einen praktischen Einsatz gut eignen. Die Netzarchitektur Long Short-Term Memory (LSTM) wurde z.B. gezielt konstruiert, um dem Effekt des „fading gradient“ zu begegnen. Dieser Effekt verhindert das praktikable Training rekurrenter Netze mittels Gradientenabstieg.

Die Netze verarbeiten die Daten in fester Durchlaufrichtung. Ist eine zu lernende Ausgabe an der konkreten Stelle aber von nachfolgenden Punkten abhängig, kann dieser Sachverhalt nicht gelernt werden. Der Kompromiss, ein Fenster statischer Größe aus Eingabedaten zu verwendet, lässt sich für rekursive Netze nur schwer einsetzen.

Es gibt daher nicht-kausale Netzwerkarchitekturen, die den Kontext, also auch punktuelle Nachfolger, berücksichtigen. Weiterhin wurden bidirektionale rekurrente Netze (BRN) definiert, die eine bereits gegebene Netzarchitektur verwenden und die Sequenz in zwei Durchlaufrichtungen gleichzeitig verarbeiten. Kontextuelle Netze erfordern Einschränkungen an ihre interne Struktur. Beide Netzarchitekturen, kontextuelle und bidirektionale, lassen die Form der Eingabedaten unangetastet und erhalten den sequentiellen Charakter der Datenverarbeitung.

In dieser Arbeit wird gezeigt, dass eine Sequenz derart in Baumstrukturen abgebildet werden kann, dass ein rekurrentes Elman-BRN auf der Sequenz dasselbe leistet wie ein rekursives Elman-Netz (auch: Simple Recurrent Network) auf den Baumstrukturen. Diese Sequenz-zu-Baum-Abbildung wird auf Baumstrukturen verallgemeinert, sodass

auch sie bidirektional restrukturiert werden können. Diese Restrukturierung wird als Form-bezogene Vorverarbeitung der Eingabedaten interpretiert.

Es werden neue Restrukturierungsverfahren definiert, also Algorithmen zur Abbildung sequentieller Daten in Baumstrukturen. Das Resultat ist unter anderem ein schnelles Verfahren zur Klassifikation translationsinvarianter Sequenzen. Weiterhin ergibt sich die Möglichkeit, eine nicht-kausale Sequenz-zu-Sequenz-Abbildung zu definieren, die unter gewissen Umständen invertierbar ist. Ein sehr einfach zu implementierendes Verfahren wird vorgestellt. Dieses verwirklicht das Konzept des „teile und herrsche“ und wird zusätzlich mit der bidirektionalen Restrukturierung kombiniert.

Alle vorgestellten Verfahren werden anhand verschiedener Klassifikationsprobleme mit dem rekurrenten Standard, basierend auf LSTM und Elman-Netzen, verglichen. Dazu werden Netze mit nur drei bis fünf Neuronen trainiert. Um ein breites Spektrum an Verwendungsszenarien abzudecken, werden synthetische und Real-world-Daten von diskreter und kontinuierlicher Natur als Eingabedaten verwendet. Die Güte des Trainings wird untereinander verglichen. Für Datensätze mit unausgewogenem Verhältnis zwischen positiven und negativen Mustern wird eine automatisch ausbalancierende Variante des Gradientenabstiegs vorgestellt. Weiterhin wird eine spezielle Initialisierung für das Trainingsverfahren Resilient Backpropagation angegeben.

Es stellt sich heraus, dass die Restrukturierungsverfahren den rekurrenten Standard übertreffen und auch dort erfolgreich sein können, wo rekurrente Netze fehlschlagen, und sie daher unbedingt zwecks Optimierung in Betracht gezogen werden sollten.

Abstract

Artificial neural networks are used in the field of machine learning to build functions that emulate expert knowledge. Feedforward networks can map between data with fixed structure, recurrent networks can emulate sequential data such as time series. Recursive networks are used for structural data such as chemical structural formulas.

Training, that is adapting the free parameters, of those nets is mostly difficult in practise. Therefore, it is amongst other things a permanent subject of research to develop special network architectures that are well suited for practical use. The network architecture LSTM for example was designed specifically to face the fading gradient which effectively renders training of recurrent networks virtually impossible by gradient descent.

The network processes the data in one fixed direction. But if a learning task requires the output for a specific point to depend on the following point, this task cannot be learned. The compromise of using an input window of static size is difficult to implement for recursive networks.

Therefore non-causal network architectures exist that take the context into account, which means they include input from successors. Furthermore, bidirectional recurrent networks (BRN) were defined using an already given network architecture to process a sequence in two directions simultaneously. Contextual networks require constraints on their internal structure. Both network architectures, contextual and bidirectional, keep the form of the input data and maintain the sequential nature of the processing.

In this work it is shown that a sequence can be mapped to tree structures such that a recurrent Elman-BRN on the sequence does the same job as a recursive Elman net (also: Simple Recurrent Network) on the tree structures. This sequence-to-tree mapping is generalised onto tree structures, so that they can be restructured bidirectionally. This restructuring is interpreted as a form-based preprocessing of the input data.

Novel methods of restructuring are defined, that is, algorithms for mapping sequences to trees. One result is a computationally efficient method for the classification of translation invariant sequences. Furthermore, the possibility to define a non-causal sequence-to-sequence mapping is concluded, which is invertible under certain conditions. One method is presented that is very easy to implement and realises the concept of Divide and Conquer. This is also combined with bidirectional restructuring.

All presented methods are compared against the recurrent default method basing on LSTM and Elman networks by learning different classification problems. Networks with only three to five neurons are used. To cover a wide range of usage scenarios, synthetic and real-world data of symbolic and continuous nature are used as input data. The quality of training is compared amongst the methods. For pattern sets with an unbalanced ratio between positive and negative patterns an auto-balancing variant of gradient descent is presented. Furthermore, a special initialisation for the training method Resilient Backpropagation is specified.

It turns out that the restructuring methods outperform the recurrent default and can be successful even where recurrent networks fail, and they should therefore be considered essential for optimisation.

Contents

List of Figures

1 Introduction

The field of machine learning summarises the efforts to emulate expert knowledge with machines. Within this field, roughly four different approaches can be made out. There are symbolic methods that, for example, derive decisions trees. There is reinforcement learning that guides a system towards the expected behaviour by rating the outputs of the system towards good or bad. Through unsupervised learning a reduced representation of the input is achieved in order to visualise or understand high-dimensional input data. Within the field of supervised learning, input and output data are explicitly specified in order to have the machine learn to map from input to output.

1.1 Supervised learning

Put shortly, supervised learning has the aim to emulate mathematical functions. Numerical methods do exist that can easily create input to output mappings that accurately represent the data at the given points. This can be achieved by interpolating polynomials, for example. These however can strongly vary upon data that is acquired through measurement errors. Furthermore, they always have unbounded monotonic (polynomial) behaviour outside the borders of the input data. As a conclusion, extrapolation is not possible. This can be overcome by methods of non-linear regression tasks. They however can only be applied if the data underlies a parametric formula whose structure is already known; the regression task only computes the parameters, the typecasting of the formula itself is not done by the machine.

Artificial neural networks can be used to approximate a function by learning on example data that can be under influence of measuring errors but also, to some degree, be evaluated outside the borders of the data that has been learned on, which is called generalisation. They can be applied on a variety of data: For fixed structures (i.e. vectors), feedforward nets can be used ([Son92]). On sequences of vectors, for example genome sequences or time series (i.e. structures with dynamic length), recurrent nets are applied ([HB05]) for example to perform speech recognition.

Recursive nets operate on tree structures that can be derived for example from chemical structural formulas ([SS97], [SG98]) in order to map drugs to chemical activity.

Upcoming with recurrent networks the problem of fading gradient occurs ([BSF94]) which often renders the training of recurrent networks impossible. Though special training methods or specific network architectures ([HS97]) can be applied, convergence problems must remain at least for hard problems ([KP90]). Preprocessing up to high mathematical conditioning ([Log00]) is done to ease the training process.

Furthermore, since the introduction of recurrent networks, a processing order is imposed upon the input structure which restricts the network to a *causal* behaviour: the nets state depends only on the inputs presented so far. For recurrent nets, often an input window is used that directly reveals a local past/future view of the input data. This window is a feedforward part of the recurrent net and, as a conclusion, must have a fixed width. This fixed width window is a compromise to the recurrent approach in order to directly access a certain context. The context is roughly considered the part of the input on which the neural net in its current state is functionally depending. Because the context seems to be important in many cases, architectures have been proposed to produce a larger context by processing input sequences in both directions ([SP97], [Bal+01], [GS05]).

In contrast to applying a certain architecture on a sequence in both directions, the connectionist approach of connections that access future activations of a certain neuron has been introduced. This approach has the drawback that functional cycles must be prevented through restrictions to the architecture. These restrictions are met by generalising the recurrent cascade correlation architecture to the Contextual Cascade Correlation architecture ([MSS04]) which is applicable also for tree structures.

The contextual approach and the bidirectional approach both create a certain context to the recurrent net. The bidirectional approach evidently keeps a sequential processing but potentially presents (iterated computations of) the whole input sequence into a single node. The contextual approach on the other hand can be generalised for structures but the context size is limited by the amount of neurons used: using n neurons results in a context size of $n - 1$ steps into future nodes.

1.2 Objectives and contributions

The objective of this thesis is to show that certain developments in the fields of neural networks applied in supervised learning can equally be approached by formal

preprocessing, that is, a processing that does not base on analysing the given input data. This is done by showing bidirectional recurrent nets being equivalent to recursive nets on certain trees. In order to be suitable for a classification task, the bidirectional restructuring is enhanced and generalised onto trees. For doing so, an asymmetrical convolution product between trees is defined. As a result, tree structures in general can be bidirectionally restructured. The approach in this thesis does not interfere with content-related preprocessing which therefore could still be applied and also no restrictions to the neural network architecture are imposed.

Throughout this thesis an unambiguous display of recursive nets will be used with certain - unambiguous - abbreviations that makes the display more clearly. The (Contextual) Cascade Correlation architecture is used to highlight the difference between pointing to data and pointing to neurons.

Several new methods of restructuring are introduced that have specific benefits. One can classify each representative of a translation invariant pattern to the same target without repeating the computation; it is also shown that it can be an invertible sequence-to-sequence mapping. One method is combined with the generalisation of bidirectional restructuring and algorithms are specified for all restructuring methods in order to spare the user to temporarily create tree structures from the input sequence.

All methods will be evaluated through several tests and compared with the recurrent default processing. They will be evaluated on long sequences consisting of synthetic or real-world data sets of discrete or continuous nature. For this, settings are defined in order to create a training environment which is the same for learning with the recurrent default and for learning with the recursive nets on the restructured sequences. Under each setting, the training will be repeated several times with different, randomised initial weights for the neural nets to start with. In most cases, two different network architectures are used, Elman nets and LSTM. This is done in order to show that long-term dependencies that might occur in the input data are less hindering for learning when using restructured inputs. When using a recursive net with the same hidden layers size as a recurrent net, the amount of adaptable free parameters (weights) still increases. To accommodate this fact, the size of each nets hidden layer is also varied in many cases in order to create an admissible comparison between the novel restructuring methods and the recurrent default.

One training data set is unbalanced according to the amount of pattern belonging to the positive and negative classes. For this data set, a balanced gradient will be defined that increases the numerical impact for patterns belonging to a underrepresented class. It turns out that for each data set the recurrent default is outperformed by at least one restructuring method regarding the training or

generalisation error (both at the same time in many cases). One of the restructuring method is shown to be sensitive to the kind of activation function used for the recursive net: while using the standard sigmoidal function it failed most of the time, using a centered sigmoidal function it succeeded. Another method turns out to be very robust and results in quick convergence towards a very low training error and significant overfitting. For most of the tests the training procedure is run for a fixed number of iterations. In order to compare the generalisation capability of the recursive nets (basing on the restructured data) against the generalisation capability of the recurrent net, some tests will be stopped when reaching a certain stopping criterion that is defined basing on experience. The stopping criterion only depends on the mean squared error and is the same for all tests under the same setting which thereby creates the frame conditions. For performing efficient gradient descent, Resilient Backpropagation will be used with a slight modification that allows to start the training with a local behaviour.

It is not the aim of this thesis to establish theoretical statements about the functions of recursive networks being able to emulate functions of recurrent networks by operating on restructured input. When not restricting the network architectures sizes to be equal it is for sure that they can emulate each other when restricting to input data of limited size due to both being universal approximators (compare [Ham99]). This is however a theoretical result which does not eliminate the possibility that for a specific task, achieved by a recurrent net, the recursive net operating on restructured input would, for example, require a much bigger hidden layer in order to achieve the same task.

1.3 Organisation of this thesis

This thesis is structured as follows: at first, the mathematical definitions are assembled in order to be able to denote vector arithmetic and structures. The shift operator is defined in order to describe mapping functions that result from a recursive net. Recursive nets are described in a very general way, including multiplicative, feedforward and recursive connections wherein each connection type is defined as a relation. Afterwards, a mapping function is assigned to a recursive net that constitutes the mapping from input to output.

Two commonly used methods for training a recursive net are described in chapter 3. The formulas are adapted towards the general recursive net used in this thesis. That is, the mix of feedforward and multiplicative connections together with recursive connections is reflected properly in the formulas. Some optimisation for certain network architectures are examined. Two different strategies for learning, basing on

the gradient information, are described. The chapter is concluded with a description of the training methods used for the experiments described in this thesis.

Afterwards, chapter 4 contains examples for neural networks that are depicted by graph-like drawings. These can be assigned to the formal definition of a recursive net by following the rules specified in this chapter. The depicted network architectures are used for training or as a motivation towards data restructuring. This chapter also includes theorems to proof the general recursive net being not too general in the sense that estimation about their expressiveness could not be applied anymore. The architecture LSTM is described separately in chapter 5 to account for a precise description and its specific, efficient training algorithm.

In chapter 6 the main contribution of this work can be found. After stating the motivation, some existing work is described. Bidirectional nets are emulated by forming a sequence into a set of trees on which Elman nets can be used. Bidirectional restructuring is defined in order to be suited for classification tasks and this process is generalised onto trees. The generalisation bases on an asymmetrical convolution product between trees. Novel methods for restructuring sequences are proposed and some features and the topic of translation invariance are dealt with. The invertibility of the periodic mode is examined. Then, algorithms are proposed that directly operate on a sequence with the transition function of a recursive net without temporarily creating the restructured sequence.

A selection of scientific literature that can be related to this thesis is found in chapter 7 and the experimental analysis of the proposed, novel restructuring methods is contained in chapter 8. Therein, different kinds of classification tasks are used to train recursive nets on restructured and unmodified sequences. The training processes are defined and their results depicted, described and interpreted.

The thesis is concluded with an overview of the benefits, applications and possible enhancements of restructuring as a formal preprocessing.

2 Definitions

In this chapter the prerequisites for describing neural network architectures are assembled, for example relations that are used to describe their internal structure. The mathematical definitions used for describing structures (sequences, trees) are specified; trees are formally defined as a recursive expression instead of a special case of directed or undirected graph. Recursive nets are formally defined as tuples as basis for feedforward nets and their biased counterparts. For the nets a net function is defined that maps a structure over a real-valued vector space into a real-valued vector space.

2.1 General

Definition 1. If not stated otherwise, a function is considered a total mapping. $\emptyset$ denotes the empty set. For a set M the set of all subsets of M is denoted as $\mathcal{P}(M)$. For nonempty sets A and B

$$B^A$$

is the *set of all functions from A to B*. For $f \in B^A, g \in C^B$ the term $g \circ f$ denotes the composition of f and g, $(g \circ f)(x) = g(f(x))$. For a nonempty set B and $A \subseteq B$ the *characteristic function of A* is defined as

$$1_A : B \to \mathbb{R},$$
$$b \mapsto 1_A(b) = \begin{cases} 1 & \text{if } b \in A, \\ 0 & \text{else.} \end{cases}$$

The expression $\exists!$ means "there exists a unique" in the sense that

$$\exists! X : E(X) \quad :\Leftrightarrow \quad \exists X : E(X) \wedge \forall Y : Y \neq X \Rightarrow \neg E(Y).$$

Vectors are considered column vectors and $x^\top$ denotes the transposition of x: $(0,1,2)^\top \in \mathbb{R}^3$. For any $x \in \mathbb{R}^m$ and $1 \leq p \in \mathbb{R}$ the expression $\|x\|_p$ is the p-norm of x and $\|x\| := \|x\|_2$.

Definition 2 (series, sequence)**.** Let $\mathbb{N} = \{1, 2, \ldots\}$, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. For a given set Σ *the set of all series over* Σ is $\Sigma^\mathbb{N}$ having $a = (a_i)_{i\in\mathbb{N}} \in \Sigma^\mathbb{N}$. Given some $n \in \mathbb{N}$, the expression $(a(i))_{i=1}^n, a(i) \in \Sigma$ respectively $(a_i)_{i=1}^n, a_i \in \Sigma$ is called a *sequence over* Σ and the *set of all sequences over* Σ is denoted as

$$\Sigma^+.$$

It is $\Sigma^+ = \bigcup_{i\in\mathbb{N}} \Sigma^i$. For any $a \in \Sigma^+$ with $a = (a_i)_{i=1}^n$ the projection to a certain index i is written by $(a)_i := a_i$. The vector concatenation is denoted with ";" so that from $x \in \mathbb{R}^m$ and $y \in \mathbb{R}^n$ follows

$$x; y = ((x)_1, \ldots, (x)_m, (y)_1, \ldots, (y)_n)^\top \in \mathbb{R}^{m+n}.$$

Definition 3 (vector and matrix product, inner product)**.** For products between vectors and/or matrices, the operator symbol is usually suppressed, but when the symbol "$\cdot$" is used, it is supposed to have weaker binding than the ";" operator, so that $Ma; b$ is the concatenation of $M \cdot a$ and b, but $M \cdot a; b$ is the product between M and $a; b$.

For $a, b \in \mathbb{R}^n$ the expression $a \overset{pw.}{\cdot} b$ is the element-wise product $a \overset{pw.}{\cdot} b = (a_i b_i) \in \mathbb{R}^n$.

For $a, b \in \mathbb{R}^n$ the term $a \bullet b$ is defined as $a^\top \cdot b \in \mathbb{R}$ as the standard scalar product. It is used to avoid the transposition operator or to show that this inner product originates from a recursive connection between neurons.

Occasionally and for ease of reading, an arrow $\vec{x}$ denotes a variable x to be a vector. This is mostly used as accentuation or in expressions containing many scalars.

Definition 4 (relation, transitive hull)**.** Let M be a nonempty set and $R \subseteq M \times M$, then R is called a *relation over M*. The statement $(a, b) \in R$ can be denoted as aRb. The *transitive hull* of R is defined as

$$R^+ := \{(a, c_n) : \exists n \in \mathbb{N} : \exists c_0, \ldots, c_n \in M : c_0 = a, (c_{i-1}, c_i) \in R \; \forall 1 \leq i \leq n\}.$$

For a relation $R \subseteq M \times M$, $A \subseteq M$ and $b \in M$ let:

$$_AR := \{m \in M : \exists a \in A : (a, m) \in R\}, \quad {}_bR := {}_{\{b\}}R,$$
$$R_A := \{m \in M : \exists a \in A : (m, a) \in R\}, \quad R_b := R_{\{b\}}.$$

2.2 Structures

Definition 5 (tree, label, child)**.** A *k-tree* T over a nonempty set Σ with $\perp \notin \Sigma$ is recursively defined as an expression $T = t(u_1, \ldots, u_k)$ where

1. $t \in \Sigma$ and

2. $\forall 1 \leq i \leq k : u_i = \perp$ or u_i is a k-tree over Σ.

Let $\perp^k = \overbrace{\perp, \ldots, \perp}^{k \text{ times}}$. Then a k-tree $t(\perp^k)$ is called *leaf* and if k is known from context it is denoted as $t()$. The *set of all k-trees over* Σ is denoted as Σ_k^+ and $\Sigma_k^* := \Sigma_k^+ \cup \{\perp\}$. A 2-tree is called *binary tree*. The function

$$\begin{aligned} \lambda :& \Sigma_k^+ \to \Sigma, \\ & T = t(u_1, \ldots, u_k) \mapsto \lambda(T) = t \end{aligned}$$

is called the *label function* and $\lambda(T)$ is the *root-label (label) of* T. Hence, for $T \in \left(\mathbb{R}^l\right)_k^+$ the expression $(\lambda(T))_i$ is defined for $1 \leq i \leq l$ with $(\lambda(T))_i \in \mathbb{R}$. The functions

$$\begin{aligned} \chi_i :& \Sigma_k^+ \to \Sigma_k^* \\ & T = t(u_1, \ldots, u_k) \mapsto \chi_i(T) = u_i \end{aligned}$$

are called *child functions* for $1 \leq i \leq k$, $\chi_i(T)$ is the *i-th child of* T and i is the *position* of the child. For $p = (i_1, \ldots, i_j)$ with $j \in \mathbb{N}$, $k \geq i_1, \ldots, i_j \in \mathbb{N}$ define

$$\chi_p = \chi_{(i_1,\ldots,i_j)} := \chi_{i_j} \circ \ldots \circ \chi_{i_1}.$$

Definition 6 (subtree, childtree, path, depth, parent, skeleton)**.** Let $T \in \Sigma_k^+$. If $j \in \mathbb{N}$ and $i_1, \ldots, i_j \in \mathbb{N}$ exists so that

$$U = \chi_{i_j} \circ \ldots \circ \chi_{i_1}(T)$$

is defined and $U \neq \perp$, then U is called *subtree* of T, the sequence $p = (i_1, \ldots, i_j)$ is a *path of* U *(within* T*)* and j is the *depth of* U *(within* T*)* and T is called *a parent* of U. This statement is abbreviated by

$$U \overset{(i_1,\ldots,i_j)}{<} T$$

and $U \leq T :\Leftrightarrow U = T \vee \exists p : U \overset{p}{<} T$. If the length of the sequence p is 1, then T is called *the parent* of U and U is also called *childtree* of T.

If $d = max\{j|\exists U \in \Sigma_k^+ : U$ is subtree of T with depth $j\}$ exists, then d is called the *depth of* T and T is said to be *finite*.

The set of all paths within T is called the *skeleton of* T denoted by

$$\text{skel}(T) := \{p|\exists U : U \overset{p}{<} T\}.$$

Example 1. The binary tree $T = a\,(\bot, b\,(c(), \bot))$ over $\Sigma = \{a, b, c, d\}$ is shown in figure 2.1. Though to every depth there exists at most one subtree, it is a binary tree due to the specified positions.

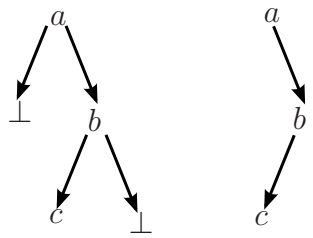

Figure 2.1. *Example for a binary tree. Left: using the symbol* $\bot$ *to determine the position of the children explicitly. Right: suppressing the symbol. If not explicitly mentioned as a binary tree, the picture to the right could also describe a* 3*-ary tree where the position* 2 *is always empty.*

Remark 1. This definition of trees can be seen as an anonymous view on directed ordered acyclic graphs with a maximum indegree of one. Since no sets of edges and vertices are defined, questions about isomorphic structures cannot be raised. Here, the tree is denoted solely by the partial ordering of its labels.

2.3 Shift operator

Definition 7 (shift operator)**.** Let $R \neq \emptyset$ and a function $F : (\Sigma)_k^* \to R$. The shift operator is defined as

$$\begin{aligned} q^{-1} :& R^{\Sigma_k^+} \to \left(R^{\Sigma_k^*}\right)^k, \\ & F \mapsto (F \circ \chi_1, \ldots, F \circ \chi_k)^\top . \end{aligned}$$

Therefore holds

$$q^{-1}\left(F\left(t(u_1, \ldots, u_k)\right)\right) = (F(u_1), \ldots, F(u_k))^\top$$

for $t \in \Sigma, u_i \in \Sigma_k^* = \Sigma_k^+ \cup \{\bot\}$. For a given position j, the corresponding shift operator is

$$\begin{aligned} q_j^{-1} :& R^{\Sigma_k^+} \to R^{\Sigma_k^*}, \\ & F \mapsto F \circ \chi_j. \end{aligned}$$

Remark 2. The shift operator maps a function to a vector of functions that applies the original function to the children of the argument. This allows to define recursive neural nets as "causal supersource transducers" as defined in [HMS07] (compare chapter 3).

2.4 Recursive neural nets

Definition 8 (general recursive net, general feedforward net)**.** Let $N = \{1, \ldots, n\} \subset \mathbb{N}$, $n = |N| \in \mathbb{N}$ (*neurons*), $k \in \mathbb{N}$ (*fan-out*), $I = \{1, \ldots, l\} \subset N$, $l = |I|$ (*input neurons*), $O \subseteq N \setminus I$ (*output neurons*), $F = \{f_i : i \in N \setminus I, f_i : \mathbb{R} \to \mathbb{R}\}$ (*activation functions*),

i) $\to \subseteq N \times (N \setminus I)$ (*feedforward connections*),

ii) $\succ\!\!\to \subseteq (N \setminus I) \times (N \setminus I)$ (*recursive connections*) and
$\mathcal{K} := \{i | \exists j : i \succ\!\!\to j\} = \{i : {}_i\!\succ\!\!\to \neq \emptyset\}$ (*context neurons*),

iii) $\overset{\pi}{\to} \subseteq \mathcal{P}(N) \times (N \setminus I)$ (*product connections*).

Further, by using a helper relation that collects all non-recursive connections,

$$\rightsquigarrow := \bigcup_{I \overset{\pi}{\to} j} \{(i,j) : i \in I\} \cup \to,$$

iv) the *acyclic* property $\forall i \in N : (i,i) \notin \rightsquigarrow^+$must apply. Further let

v) $V = (v_{ij}) \in \mathbb{R}^{n \times n}$, $v_{ij} = 0$ if not $i \to j$,
$W = (w_{ij\kappa}) \in \left(\mathbb{R}^k\right)^{n \times n}$ (*weight tensor*) with $w_{ij} := (w_{ij\kappa})_\kappa \in \mathbb{R}^k$, $W_\kappa := (w_{ij\kappa})_{ij} \in \mathbb{R}^{n \times n}$ (*layer of the weight tensor*) and $w_{ij} = 0$ if not $i \succ\!\!\to j$,
$\mathcal{W} := (V, W)$ (*weights*), and

vi) $\xi : \mathcal{K} \to \mathbb{R}$ (*initial context*).

Then the tuple $(N, I, O, F, \mathcal{W}, \to, \succ\!\!\to, \overset{\pi}{\to}, \xi)$ is called a *general recursive neural net* (or "recursive net" for short) and $(N, I, O, F, V, \to, \overset{\pi}{\to})$ is called a *general feedforward neural net* ("feedforward net"). A *recurrent net* is a recursive net with $W \in \mathbb{R}^{1 \times n \times n} = \mathbb{R}^{n \times n}$, that is, with fixed parameter $k = 1$.

Remark 3. Even though harder to read, in this thesis the connection structure within a neural net is being noted by means of relations ($\rightarrow$, $\nrightarrow$, $\overset{\pi}{\rightarrow}$) instead of graphs. This prevents ambiguity with the potential input structure to be mapped by the according net function.

Later, when different classes of neural nets are being described, graph-like drawings arranged in a bottom-up-fashion will be used such that oriented connections are drawn as arrows pointing up. In opposite, examples for input structures will be drawn in a top-down-fashion, so that children are pointed downwards at and leaves are positioned at the bottom.

For $(i,j) \in \rightarrow, \nrightarrow, \overset{\pi}{\rightarrow}$ one can read "(i,j) in forward", "(i,j) in recurrent" and "(i,j) in product", respectively, and for $i \rightarrow j$, $i \nrightarrow j$ and $i \overset{\pi}{\rightarrow} j$ one can read "i before j", "i in j" and "i product j".

The name *Folding Architecture* occurs as a synonym for a recursive net.

Remark 4. When the relations of a general recursive net are not explicitly defined by a single equation but of several requirements (logical statement such as "$A \times B \subset \rightarrow$" or "$3 \nrightarrow 3$"), the relation is defined as the intersection of all (i.e. the smallest) relations fulfilling these requirements. For example, the statements "$1 \rightarrow 2$, $2 \nrightarrow 1$ and $1 \nrightarrow 1$" therefore define the relations $\rightarrow = \{(1,2)\}$, $\nrightarrow = \{(1,1),\ (2,1)\}$ and $\overset{\pi}{\rightarrow} = \emptyset$.

Definition 9 (net function)**.** Let $a = (N, I, O, F, \mathcal{W}, \rightarrow, \nrightarrow, \overset{\pi}{\rightarrow}, \xi)$ be a general recursive net and for $i \in N$ and $f_i \in F$ let

$$o_i : \left(\mathbb{R}^l\right)_k^* \rightarrow \mathbb{R}, \tag{2.1}$$
$$T \mapsto o_i(T) = \begin{cases} (\lambda(T))_i & \text{if } i \in I, \\ \xi(i) & \text{if } i \in \mathcal{K} \text{ and } T = \perp, \\ f_i(net_i(T)) & \text{else,} \end{cases}$$

be a partial function (*neuron output* or *activation*) that is undefined for $T = \perp \wedge i \notin \mathcal{K}$, and for $j \in N$ let

$$net_j : \left(\mathbb{R}^l\right)_k^+ \rightarrow \mathbb{R},$$
$$T \mapsto net_j(T) = \sum_{i \rightarrow j} v_{ij} o_i(T) + \sum_{i \nrightarrow j} w_{ij} \bullet q^{-1}(o_i(T)) + \sum_{J \overset{\pi}{\rightarrow} j} \prod_{i \in J} o_i(T)$$

be a function (*net input*). Then

$$\mathcal{F}_a : \left(\mathbb{R}^l\right)_k^+ \rightarrow \mathbb{R}^{|O|},$$
$$T \mapsto \mathcal{F}_a(T) = (o_j(T))_{j \in O}$$

is called the *net function* of a.

Let $b = (N, I, O, F, V, \to, \overset{\pi}{\to})$ be a feedforward net. By defining a recursive net $a(b) := \left(N, I, O, F, (V, \vec{0}), \to, \emptyset, \overset{\pi}{\to}, \vec{0}\right)$ the function

$$\begin{aligned}\mathcal{F}_b :&\mathbb{R}^l \to \mathbb{R}^{|O|},\\ &t \mapsto F_{a(b)}\left(t(\perp^k)\right)\end{aligned}$$

is called the *net function* of b.

Remark 5. The function o_i makes use of function net_j and vice versa, but the previous definition is well-defined because of the acyclic property. The incoming weight vector for neuron $j \in N$ is the j-th column of the weight matrix. For an appropriate (layered) connection structure $net_B(x) = V^\top o_A(x) + \ldots$ can be computed with $A, B \subset N$ and $V \in \mathbb{R}^{|A|\times|B|}$.

The expression $w_{ij} \bullet q^{-1}(o_i(T))$ represents an inner product and having $T = t(u_1, \ldots, u_k)$ it equals

$$\sum_{\kappa=1}^{k} w_{ij\kappa} q_\kappa^{-1} \circ o_i(T) = \sum_{\kappa=1}^{k} w_{ij\kappa} o_i(u_\kappa).$$

Though $\perp$ is not considered to be a tree, the neuron output function o_i assigns a value to it if $i \in \mathcal{K}$. These values are determined simply by definition of ξ, the initial context.

The values $(o_i(T))_{i\in\mathcal{K}}$ are also referred to as the *state within* T.

No further assumptions are made on ξ, even though this could be reasonable regarding a certain learning task at hand. Possible assumptions are $\xi(i) \in f_i^{-1}(\mathbb{R})$ or even the implicit statement

$$\xi = (o_i(Z))_{i\in\mathcal{K}} \quad \text{with } Z = \vec{0}() \in \left(\mathbb{R}^l\right)_k^+.$$

By definition, a connection $I \overset{\pi}{\to} j$ does not have a scalar factor (weight) within the net input of j but when defining an additional neuron l with $f_l = id$ and $I \overset{\pi}{\to} l$, a weight can be introduced using $l \to j$ as replacement for $I \overset{\pi}{\to} j$.

Obviously Σ_1^+ is isomorphic to Σ^+ regarding net functions using

$$a_n(a_{n-1}(\ldots(a_1(\perp)\ldots))) \mapsto (a_1, \ldots, a_n).$$

Definition 10 (bias, biased recursive net, biased net function)**.**
Let $\mathcal{N} = (N, I, O, F, \mathcal{W}, \rightarrow, \succcurlyeq, \xrightarrow{\pi}, \xi)$ be a recursive net and

$$\theta : N \setminus I \rightarrow \mathbb{R}, i \mapsto \theta(i) = \theta_i$$

be a function. Then θ_i is called the *bias of* i and $\mathcal{M} := (N, I, O, (F, \theta), \mathcal{W}, \rightarrow, \succcurlyeq, \xrightarrow{\pi}, \xi)$ is called a *biased recursive net*.

Given the terms from definition 9, but replacing $f_i(net_i(T))$ by $f_i(\theta_i + net_i(T))$ within equation (2.1), the resulting net function is called *θ-biased net function of* $\mathcal{N}$ or *net function of* $\mathcal{M}$. These terms and definitions transfer to feedforward nets in the obvious way.

Remark 6. A biased recursive net can be transformed into a non-biased recursive net either by providing an additional input neuron i with constant value 1, or through introducing a dummy neuron i with $f_i(x) = 1$, or, if to use a certain activation function f, by introducing a dummy neuron with one self recurrent connection and $w_{ii\kappa} := \delta_{1,\kappa}$ and defining $\xi(i)$ as non-zero fixed point of f (which then must exist). In the first two cases it is $v_{ij} = \theta_j$, in the last case $v_{ij}/\xi(i) = \theta_j$.

As concrete values of weights are to be found to adopt certain tasks, their values are usually not mentioned. In this case the recursive net mentioned actually stands for the class of all net functions that can be achieved by varying the parameters (weights, bias, initial context). The class of all nets that meets the restrictions for $\mathcal{N}$ is also called an **architecture**.

Remark 7. Activation functions used in this thesis are the *sigmoid* function $\mathrm{sgd}(x) = \frac{1}{1+e^{-x}}$ and its scaled and shifted versions $a \cdot \mathrm{sgd}(x) - a/2$ with $a > 0$; they are differentiable and can be used for gradient descent based learning methods. Other activation functions commonly used for neural nets are the *Heaviside* function $H(x) = 1$ for $x \geq 0$, $H(x) = 0$ for $x < 0$, which is used in a Hopfield net. As a symmetrical activation function, the hyperbolic tangent $tanh(x) = 2\,\mathrm{sgd}(2x) - 1$ occurs.

3 Training of recursive nets

The benefit of recurrent and recursive networks is that they process input data of dynamic length or size but an artificial neural network is of no use if its net function does not serve any specific purpose. However, even when a network architecture is known that has the capability to serve a specific task, the weights and biases must be found or approximated to unknown values that are mostly non-unique.

As the result of a training process is not unique and the according net is a black box that does not on its own reveal the reason behind its functionality, several different training processes must be conducted and each resulting net's performance must be checked on a test set that was not part of the training process. Each of this tasks requires to learn on a set of example data that is called the *pattern set*.

While specific problems can be trained with optimised and adapted strategies, the general method for learning with recurrent and recursive networks always means modifying a non-linear recursive vectorial function. In general this is done by gradient descent on an continuously differentiable error function, which will be explained in this chapter.

Pattern set

The pattern sets dealt within this thesis are of the form

$$M \subset \{(x, y) | x \in \left(\mathbb{R}^l\right)_k^+, y \in \mathbb{R}^n\}, k \in \mathbb{N}$$

where x is the input data and y the output of an unknown function that is to be simulated with a recursive net. The output dimension $n = |O|$ defines the size of the output layer. For a classification it is $n = 1$ with $0 \leq y_1 \leq 1$ and for a multi class problem this holds pointwise and optionally $\|y\|_1 = 1$ is requested so that the input data can be interpreted to belong into each class with a certain probability y_i and y is the probability distribution of this pattern.

Now the target is to find a recursive net N such that holds:

$$\mathcal{F}_N(x) \approx y \quad \forall (x, y) \in M.$$

Equality is not wanted and mostly not achieved; the closer the output values of the net stick to the training data, the more the net tends to interpolate and loose the ability to simulate the behaviour of the unknown function that underlies the pattern set M. This is called "overfitting".

Other forms of pattern sets; structural transducers

The ability to apply artificial neural networks for machine learning on structured data (trees) was introduced after their application to sequential data. While the formalisation of a pattern set for classification is obvious even for structured inputs, the question arises how neural nets can process data in general. In [HMS07] this question is approached by specifying several definitions for structure-to-structure mappings. In this context, the net function of a recursive net represents a "causal supersource transducer". When the net functions output is saved not only for the root of the input tree but also for every subtree, an output structure with the same skeleton as the input structure can be constructed. For a sequence, this equals a time series that maps a sequence of input vectors to a sequence of state/output vectors. In general, the resulting function is a "causal I/O-isomorphic transducer" which maps an input structure to an output structure with computed labels. It can be trained on a pattern set

$$M \subset \{(x,y)|x \in \left(\mathbb{R}^l\right)_k^+, y \in \left(\mathbb{R}^n\right)_k^+, \mathrm{skel}(x) = \mathrm{skel}(y)\}, k \in \mathbb{N}.$$

When the output values for parts of the input structure are of no interest, the term "error injection" is common through the literature. For the parts of the input where the output is of relevance, an error is said to be "injected" during training. This notion is transferred from the error injection of the output layer of a feedforward network during backpropagation (see below). Having error injection only in the root (supersource) reduces an I/O-isomorphic transducer to the equivalent of a supersource transducer.

3.1 Training error

To measure how close a given net approximates the training data, an error function $E = E(M)$ is required. This is usually defined over a single pattern $E(m)$ and defining $E(M) := \Sigma_{m \in M} E(m)$. Let the pattern be $m = (x, y)$ and $f = \mathcal{F}_N$ the net

function of a recursive net N. One of the error functions used in this thesis is the Sum of Squared Error (SSE):

$$E_{\text{SSE}} := \frac{1}{2}\|f(x) - y\|^2 = \frac{1}{2}\sum_{i=1}^{l}((f(x))_i - y_i)^2.$$

It is mostly used for training on patterns with continuously distributed outputs and simply implements the training of a recursive net as a non-linear regression task that searches for minima on a scalar field.

Another error function is the cross entropy (CE) based function defined for $n = 1$ (i.e. $y_1 = y$) and pattern with $y \in \{0, 1\}$ by

$$E_{\text{CE}} := -\sum y_j \ln o_j(x) + (1 - y_j)\ln(1 - o_j(x))$$

which is mostly used for binary classification tasks. This error function has been proposed in [JS98] based on stochastic reasoning. However, also the SSE function can be used for classification.

An error function maps the net output and the target vector to a scalar value $E \in \mathbb{R}$ and thereby defines a scalar field over the free parameters of the underlying net (through the net functions output) and the input pattern $m = (x, y)$.

3.2 Gradient based

As the net function is a non-linear recursive vectorial function, analytical methods for finding a global minimum usually do not exist. Instead, the error function defines a scalar field on which minima can be searched with gradient descent. After calculating the gradient, the parameters (weights) of the net can be modified in the negative direction of this gradient and the process can be repeated. For calculating the gradient, two different algorithms can be used that differ in their memory and computation complexity. They use different variables ("error signals") that are stored in memory either while processing a pattern or after computing the net's output. One of the algorithms base on the idea of unfolding the recursive net for a given structure into a feedforward net by copying the weights accordingly. Both use a recursive formula to compute a target output not only for output neurons. Therefore the chain rule for differentiating functions in $\mathbb{R}^n$ shall be mentioned:

The chain rule

Let $f : \mathbb{R}^a \to \mathbb{R}^b$ and $g : \mathbb{R}^b \to \mathbb{R}^c$ be two functions and d denote the total derivative operator, that is, the operator that maps a function h and a point x to the linear function $dh|_x$ that approximates h within an open neighbourhood of x. Then if all functions are totally differentiable it is

$$d(g \circ f)|_x = dg|_{f(x)} \circ df|_x.$$

The expression dg does not represent a matrix but can be implemented over the standard basis as the Jacobian matrix of dg and denoted by ∂g. Using this rule one can introduce a helper function $f(w) := (w, \ldots, w)$ that copies one argument $w \in \mathbb{R}$ multiple times. Differentiating a function $g^*(w)$ that uses multiple copies $w(t)$ of w through a recursive definition, for example a recursive net, can now be understood as differentiating a function $g \circ f(w)$ and the chain rule yields $\partial g^*(w)/\partial w = dg|_{w(1),\ldots,w(t)} \circ df(w)|_w = \partial g \cdot \partial f = \sum_t \partial g/\partial w(t)$.

The gradient

For the training of a recursive net the error function E is thought of as a function only of the free parameters of the net wherein a potential bias is assumed to be modelled as an additional weight. The gradient is then

$$\partial E := (\partial E/\partial u)$$

with u spanning over all weights of the net. Calculating the gradient is the first task when training a net and this can be done in mainly two different ways: Backpropagation Through Time and Real Time Recurrent Learning. Both have been introduced for recurrent nets. The actual complication in calculating the gradient results from the recursive definition of the net function. Even for a feedforward net, for a connection $i \to j$ all neuron outputs o_k with $k \in {}_j \rightsquigarrow^+$ are a function of v_{ij} so a formula is required to efficiently calculate $\partial E/\partial u$ with $u = v_{ij}$. This is constituted by the so-called *Backpropagation*.

3.2.1 Backpropagation

Given a feedforward net with neuron set N and net function f with feedforward weight matrix W_1. Let $m = (x, y) \in \mathbb{R}^l \times \mathbb{R}^n$ be a pattern and $E = E(m)$. For

Backpropagation, the chain rule is used in two ways: at first, $dE|_{W_1}$ is understood as

$$dE(W)|_{W_1} = dE(O)|_{O(W_1)} \circ dO(W)|_{W_1}$$

with O being the variables for the output neurons activations and W_1 the actual point in the weight space at which the gradient is calculated. It is important to slice the error function at O to be able to specify a recursion formula for error *signals* using a non-recursive formula for error *injection* because output neurons o_i, o_j could be functionally dependent by $i \to j$ for example. The injected error is defined as the Jacobian matrix $\partial E/\partial O = (e_j)_{j \in O}$ of $E(O)$ by

$$e_j := \frac{\partial E(O)}{\partial o_j}.$$

For $E_{\mathrm{SSE}}(x) = \|f(x) - y\|^2/2$ the formulas are

$$e_j(x) = o_j(x) - y_j \ \forall j \in O$$

and for $E_{\mathrm{CE}}(x) = -\sum_{j \in O} y_j \ln o_j(x) - (1 - y_j) \ln(1 - o_j(x))$ they are

$$e_j(x) := -\frac{y_j - o_j(x)}{o_j(x)(1 - o_j(x))} \ \forall j \in O.$$

Using these and the chain rule as mentioned above and having w_{ij} any component of W_1 results in

$$\partial E/\partial w_{ij} = \sum_{k \in O} e_k \frac{\partial o_k}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

The actual Backpropagation is to define *error signals* by

$$\vartheta_j := \sum_{k \in O} e_k \frac{\partial o_k}{\partial net_j}, \quad j \in N$$

and using the chain rule with respect to all neurons receiving incoming connections from j (and thus directly being a function of it) to specify a recursion formula. Regarding only the $\to$ relation the formulas can be deduced from $\partial/\partial net_j = \sum_{j \to l} \partial/\partial net_l \cdot \partial net_l/\partial net_j$.

Let $f_i'(x) = \partial f_i(x)/\partial x$, for $L \subseteq N$ let $o_L(T) := \prod_{m \in L} o_m(T)$ and expand $e_j := 0$ for $j \notin O$. The recursion formulas can then be expressed by

$$\vartheta_j = f_j'(net_j(x)) \left[e_j(x) + \sum_{j \to k} w_{jk} \vartheta_k + \sum_{\substack{J \overset{\pi}{\to} k \\ j \in J}} \vartheta_k o_{J \setminus \{j\}}(x) \right].$$

This error signal is computed by propagating back the error signals of functionally dependent neurons and, if specified, injecting the error of the neuron, which is the outer derivative of $E = E(O)$. The result is pointwise scaled by f_j'. The recursion starts at neurons without outgoing connections. Using these error signals, the gradient can be computed as

$$\partial E/\partial v_{ij} = \vartheta_j o_i(x).$$

This technique can be transferred to recursive nets as done by BPTT in the next section.

A feedforward net can have a structure in layers $L_0, \ldots, L_{h+1}$ where L_{h+1} is the distinct output layer, $\overset{\pi}{\to} = \emptyset$ (i.e. a MLP, compare chapter 4.3) with weight matrices $V_1, \ldots, V_{h+1}$ between the layers. Fur such a net the recursion can be made explicit by $\vartheta_O = f_O'(net_O(x)) \overset{pw.}{\cdot} e_O$ for the output layer $L_{h+1} = O$ and recursively $\vartheta_{L_n} = f_{L_n}'(net_{L_n}(x)) \overset{pw.}{\cdot} \left(V_{n+1} \vartheta_{L_{n+1}}\right)$.

3.2.2 Backpropagation Through Time/Structure

Backpropagation Through Structure (BPTS) and Backpropagation Through Time (BPTT) consider the recursive net unfolded into a feedforward net by copying the weights and using the chain rule by meaning of $\partial/\partial w = \sum \partial/\partial w(t)$ to determine a recursive formula for error signals. BPTS has been described in [GK96] as a generalisation of BPTT as described for example in [Wer90]. BPTT as introduced in the literature is based on recurrent networks without feedforward connections (the recurrent net in [Wer90] actually has another weight matrix for input not only from $t-1$ but also from $t-2$, thus reaching over two time steps), so error backpropagation through a feedforward part of the net is not required. As a conclusion, for the general recursive net, enhanced formulas are required to reflect the sums over $\to$ and $\overset{\pi}{\to}$ within the net input properly. Though neither method is used for the experiments conducted in this thesis, they constitute a time-efficient and broadly used method for computing the gradient and will be shortly described for nets with weights v_{ij} and w_{ijk}, net function f and output layer O in the following.

For a pattern $m = (T, y) \in \left(\mathbb{R}^l\right)_k^+ \times \mathbb{R}^n$ the error signals

$$\vartheta_i(U) := \sum_{k \in O} e_k \frac{\partial o_k(T)}{\partial net_i(U)} \quad \forall U \leq T, i \in N$$

are defined. Using $e_j = e_j(T)$ from feedforward Backpropagation with $x = T$, the error signals can be computed according to the following recursion formulas:

$$\vartheta_i(T) = f_i'(net_i(T)) \left[\sum_{i \to j} v_{ij} \vartheta_j(T) + \sum_{\substack{I \xrightarrow{\pi} j \\ i \in I}} \vartheta_j(T) o_{I \setminus \{i\}}(T) + e_i(T) \right],$$

$$\vartheta_i(U) = f_i'(net_i(U)) \left[\sum_{i \to j} v_{ij} \vartheta_j(U) + \sum_{\substack{I \xrightarrow{\pi} j \\ i \in I}} \vartheta_j(U) o_{I \setminus \{i\}}(U) + \sum_{i \overset{k}{\rightarrowtail} j} w_{ijk} \vartheta_j(U^*) \right]$$
$$\forall U \overset{(k)}{<} U^* \leq T,$$

where U is a childtree of U^* within T at position k. For an I/O-isomorphic transducer, the summand $e_i(U)$ must be included in every $\vartheta_i(U)$. For a recursive MLP (page 42), vector arithmetic could be exploited again to write the recursion formula more compact with an evident order of recursion. The gradient can now be computed as

$$\partial E(T)/\partial v_{ij} = \sum_{U \leq T} \vartheta_j(U) o_i(U),$$
$$\partial E(T)/\partial w_{ijk} = \sum_{U \leq T} \vartheta_j(U) o_i(\chi_k(U)).$$

Given an input structure S of size $T = 1 + |\operatorname{skel}(S)|$ respectively a sequence of length T, a recursive net with neuron set of size N and W weights (wherein product connections count according to the number of factors). Then BPTT has memory complexity $\mathcal{O}(NT)$, as it requires storing all neuron activations within each copy of the net, and computational complexity $\mathcal{O}(WT)$. This transfers to BPTS where the maximum fanout k exists as a linear factor in W.

3.2.3 Real Time Recurrent/Recursive Learning

Real Time Recurrent Learning (RTRL) has been proposed in [WZ89] as a learning scheme that does not require to store the pattern entirely in memory, which renders it able to process patterns of arbitrary length.

The following error signals are defined:

$$\begin{aligned}\vartheta_{ij\kappa n}(T) :=& \partial o_n(T)/\partial w_{ij\kappa} && \forall 1 \leq \kappa \leq k, i \succcurlyeq j, n \in N \setminus I,\\ \vartheta_{ijn}(T) :=& \partial o_n(T)/\partial v_{ij} && \forall i \to j, n \in N \setminus I.\end{aligned}$$

Together with the net output they are being updated directly by differentiating the output function into every possible weight. Remark that $o_{L\setminus\{l\}}(T) = o_L(T)/o_l(T)$ for $o_l(T) \neq 0$. $\delta_{i,j}$ represents the Kronecker delta:

$$\begin{aligned}\vartheta_{ij\kappa n}(T) =& f_n'(net_n(T))\Big[\sum_{l\to n} v_{ln}\vartheta_{ij\kappa l}(T) + \delta_{j,n}o_n(\chi_\kappa(T))\\ &+ \sum_{l\succcurlyeq n} \vec{w}_{ln} \bullet q^{-1}\vartheta_{ij\kappa l}(T) + \sum_{L\overset{\pi}{\to} n} o_L(T)\sum_{l\in L}\vartheta_{ij\kappa l}(T)/o_l(T)\Big],\\ \vartheta_{ijn}(T) =& f_n'(net_n(T))\Big[\delta_{j,n}o_n(T) + \sum_{l\to n} v_{ln}\vartheta_{ijl}(T)\\ &+ \sum_{l\succcurlyeq n} \vec{w}_{ln} \bullet q^{-1}\vartheta_{ijl}(T) + \sum_{L\overset{\pi}{\to} n} o_L(T)\sum_{l\in L}\vartheta_{ijl}(T)/o_l(T)\Big].\end{aligned}$$

In this formula the neuron outputs are assumed to be nonzero, so the full product can be reused and must not be computed again. Using these error signals, the gradient is calculated to

$$\begin{aligned}\partial E/\partial v_{ij} &= \sum_{l\in O} e_l(T)\vartheta_{ijl}(T),\\ \partial E/\partial w_{ij\kappa} &= \sum_{l\in O} e_l(T)\vartheta_{ij\kappa l}(T).\end{aligned}$$

The update has computational complexity of $\mathcal{O}(W^+)$ when ranging over all $n \in N\setminus I$, where W^+ denotes the number of weights with $L \overset{\pi}{\to} n$ counting $|L|$ times. The total complexity for one update is $\mathcal{O}(WW^+)$ or shortly $\mathcal{O}(W^2)$ if $|L|$ is bounded by a constant. The number of updates is the size T of the input structure resulting in total computational complexity of $\mathcal{O}(TW^2)$ wherein the maximum fanout k constitutes quadratic factor. The gradient costs $\mathcal{O}(NW) \subset \mathcal{O}(W^2) \subset \mathcal{O}(TW^2)$ with N being the number of neurons. The memory complexity depends on the implementation: processing all subtrees of same depth yields $\mathcal{O}(BNW)$ having B the maximum number of subtrees at the same depth ("breadth") which is suitable for sequences where $k = B = 1$ or generally sequence-like trees. For a k-regular tree it would be $B = \mathcal{O}(T)$. Traversing a tree in post-order by starting at the left-most subtree and successively filling $log_k(T)$ different variables adding up the ϑ at each depth the memory complexity becomes $\mathcal{O}(log(T)NW)$. Compare 6.6.1 for an example. In [Sch92] ([Ham07]) a combination of RTRL with BPTT is described

which reduces the complexity from $\mathcal{O}(W^2) = \mathcal{O}(N^4)$ per time step to $\mathcal{O}(N^3)$ in average.

3.2.4 Optimised RTRL

The formulas for RTRL are inefficient because they compute ϑ even for weights that cannot have influence anywhere but in the root of the input structure T. To specify optimised formulas, the notation $u_{ij(\kappa)}$ is used to summarise feedforward and recursive weights $u = v_{ij}$ or $u = w_{ij\kappa}$ into one expression; $\vartheta_{ij(\kappa)l}$ summarises the error signals for $u_{ij(\kappa)}$. To access the activation that is carried into net input over a weight $u_{ij(\kappa)}$ the notation $T_{(\kappa)}$ is used with $T_{(\kappa)} := \chi_\kappa(T)$, if κ is specified, and $T_{(\kappa)} := T$, if not. This yields

$$\frac{\partial o_j(T)}{\partial u_{ij(\kappa)}(T)} = f_j'(net_j(T))o_i(T_{(\kappa)}).$$

Architectures with dedicated output neurons

The context neurons are those with *outgoing* recursive connections, they define the source connection points of the different copies of the recursive net when it is unfolded into a feedforward net. Assume Ω to be the set of all neurons that are behind every context neuron considering all feedforward connections using the helper relation $\rightsquigarrow$ from definition 8 by

$$\Omega := N \setminus \mathcal{K} \setminus (\rightsquigarrow_{\mathcal{K}}^{+}).$$

Hence for every weight incoming into $j \in \Omega$ it is

$$\partial o_l(T)/\partial v_{ij} = \sum_{U \leq T} \partial o_l(T)/\partial v_{ij}(U) = \partial o_l(T)/\partial v_{ij}(T)$$

which can be computed using feedforward Backpropagation and only $\vartheta_{ij(\kappa)l}$ for $l, j \notin \Omega$ must be updated. For the most neural net architectures used in practise $O \subseteq \Omega$ holds because they have a dedicated output layer. As a conclusion this optimisation can be used in general. Compare figure 3.1.

Reducing to recurrent and context neurons

The target connection points of the copied net are the neurons $\mathcal{R} := {}_{\mathcal{K}}\succ$. They are referred to as *recurrent neurons* in the following. Optimised formulas for computing

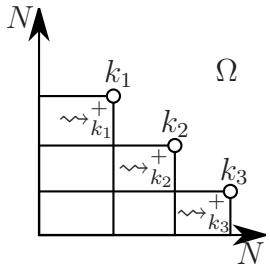

Figure 3.1. *Ω for $\mathcal{K} = \{k_1, k_2, k_3\}$. Left and below each neuron k the sets $\xrightarrow{\pi}_k$ and $\rightarrow_k$ are arranged and they recursively form the rectangular shape of $\rightsquigarrow^+_k$.*

ϑ that rely on feedforward Backpropagation can be acquired. This is done by slicing the recursion formulas at $\mathcal{K}$ and $\mathcal{R}$. Let

$$\begin{aligned} \delta_j(T) :=& \sum_{l \in O} e_l(T) \frac{\partial o_l(T)}{\partial o_j(T)} && \forall j \in N, \\ \delta_{jl}(T) :=& \frac{\partial o_l(T)}{\partial o_j(T)} && \forall l \in \mathcal{K}, j \in N \end{aligned}$$

be error signals computed via feedforward Backpropagation in the following order:

i) $\delta_j(T)$ for every $j \in \Omega \cup \mathcal{K}$,

ii) $\delta_{jl}(T)$ for all $j \in N \setminus \Omega$ and $l \in \mathcal{K}$,

iii) $\delta_j(T) = \sum_{k \in \mathcal{K}} \delta_k(T) \delta_{jk}(T)$ for the remaining $j \in N \setminus \Omega \setminus \mathcal{K}$.

Note that the expressions $\delta_{ij}(T)$ represents the error signal; the Kronecker delta $\delta_{i,j}$ is not used in these expressions. It is $\delta_{kk}(T) = 1$ but depending on $\rightarrow$ and $\xrightarrow{\pi}$ for $i \neq j$ it can be $\delta_{ij}(T) \neq 0$ if $i \rightsquigarrow^+ j$. Using these error signals, only $\vartheta_{ij(\kappa)l}$ for $l \in \mathcal{K}$ and $j \notin \Omega$ must be updated using the following formula by slicing at $\mathcal{R}$ and $\mathcal{K}$:

$$\begin{aligned} \vartheta_{ij(\kappa)l}(T) :=& \frac{\partial o_l(T)}{\partial u_{ij(\kappa)}} = \left(\partial / \partial u_{ij(\kappa)}(T) + \sum_{U<T} \partial / \partial u_{ij(\kappa)}(U) \right) o_l(T) \\ =& f'_j(net_j(T)) \delta_{jl}(T) o_i(T_{(\kappa)}) \\ &+ \sum_{m \in \mathcal{R}} \frac{\partial o_l(T)}{\partial o_m(T)} \sum_{\eta=1}^{k} \sum_{n \in \mathcal{K}} \frac{\partial o_m(T)}{\partial o_n(\chi_\eta(T))} \sum_{U \leq \chi_\eta(T)} \frac{\partial o_n(\chi_\eta(T))}{\partial u_{ij(\kappa)}} \\ =& f'_j(net_j(T)) \delta_{jl}(T) o_i(T_{(\kappa)}) \\ &+ \sum_{m \in \mathcal{R}} \delta_{ml}(T) \sum_{\eta=1}^{k} \sum_{n \in \mathcal{K}} \delta^*_{nm\eta}(T) \vartheta_{ij(\kappa)n}(\chi_\eta(T)), \end{aligned}$$

using the abbreviation $\delta^*_{ij\eta}(T) := \frac{\partial o_j(T)}{\partial o_i(\chi_\eta(T))}$ for $i \in \mathcal{K}$ and $j \in \mathcal{R}$. Using these error signals and the same slicing method the gradient can be computed as

$$\begin{aligned}\partial E/\partial u_{ij(\kappa)}(T) =& f'_j(net_j(T))\delta_j(T)o_i(T_{(\kappa)}) \\ &+ \sum_{m\in\mathcal{R}} \delta_m(T) \sum_{\eta=1}^{k} \sum_{n\in\mathcal{K}} \delta^*_{nm\eta}(T)\vartheta_{ij(\kappa)n}(\chi_\eta(T)).\end{aligned}$$

Computing δ^* is crucial and depends on the architecture that is used. For $j \in \Omega$, only the summand containing f'_j must be computed.

Isolated recurrent neurons

If $\mathcal{R}$ and $\mathcal{K}$ are sufficiently isolated regarding $\rightarrow$ and $\stackrel{\pi}{\rightarrow}$ but sufficiently connected regarding $\succ$ by means of

$$\mathcal{K} \times \mathcal{R} \subseteq \succ, \quad \rightsquigarrow^+_{\mathcal{R}} \cap \{\mathcal{K} \cup \mathcal{R}\} = \emptyset = \rightsquigarrow^+_{\mathcal{K}} \cap \mathcal{K}, \tag{3.1}$$

that is, if $\mathcal{K}$ and $\mathcal{R}$ are fully connected and no context/recurrent neuron directly or indirectly serves as feedforward input for a recurrent neuron, the δ^* reduces to a simple expression:

$$\delta^*_{ij\eta}(T) = f'_j(net_j(T))w_{ij\eta}$$

and using this the recursion and gradient formulas become

$$\begin{aligned}\vartheta_{ij(\kappa)l}(T) =& f'_j(net_j(T))\delta_{jl}(T)o_i(T_{(\kappa)}) \\ &+ \sum_{m\in\mathcal{R}} f'_m(net_m(T))\delta_{ml}(T) \sum_{n\in\mathcal{K}} \vec{w}_{nm} \bullet q^{-1}\vartheta_{ij(\kappa)n}(T), \\ \partial E/\partial u_{ij(\kappa)}(T) :=& f'_j(net_j(T))\delta_j(T)o_i(T_{(\kappa)}) \\ &+ \sum_{m\in\mathcal{R}} f'_m(net_m(T))\delta_m(T) \sum_{n\in\mathcal{K}} \vec{w}_{nm} \bullet q^{-1}\vartheta_{ij(\kappa)n}(T).\end{aligned}$$

The requirement (3.1) is met for example by a recursive MLP and by an Elman net. However, for the latter also holds $\mathcal{R} = \mathcal{K} = N \setminus I \setminus O$ because it is a very simple architecture and only the optimisation regarding Ω is effective. The net that is the result of a Recursive Cascade Correlation maximally violates the requirement (3.1), compare figure 4.4.

In 5.2.1 and following formulas are given that base on assuming only a subset of $\mathcal{R}$ being actual recurrent neurons.

3.3 What to do with the gradient?

For brevity, let W be the vector of all weights v_{ij} and w_{ijk} of a recursive net N, u denote any component of W, M be the pattern set to learn on and

$$\partial E|_W = \sum_{m \in M} \partial E(m)|_W$$

being the gradient of the error function calculated for example with the methods mentioned above.

3.3.1 Gradient descent - the standard update rule

Gradient descent is an iterative numerical method that is based on the idea that the total derivative (the gradient) $\Delta := \partial E|_x$ of a scalar field at point x is the unique locally approximating linear function within an open neighbourhood of x with ∂E being the direction of steepest ascension of this scalar field. As a conclusion, for a sufficiently small $\epsilon > 0$ it is $E(x - \epsilon\Delta) < E(x)$ if $\Delta \neq 0$.

The iteration uses a constant factor $0 < \alpha \in \mathbb{R}$, the "learning rate" (in numerical literature: "step size"). Gradient descent defines a sequence of weights by

$$W^{(i+1)} := W^{(i)} - \alpha \, \partial E|_{W^{(i)}}.$$

The iteration is started with a randomly chosen $W^{(0)}$ and stopped when the error E is sufficiently small.

The magnitude of the components of $W^{(0)}$ (i.e. the initial conditions) has been rather few subject to research. While for feedforward nets, an initial range of $-0.1 \leq u \leq 0.1$ can be well-suited, several experiments conducted in [KP90] showed that using fairly large values can be necessary because otherwise the sequence of weights does not start to converge towards any solution. This fact is reconfirmed through some experiments conducted in this thesis.

Momentum term

As gradient descent performs bad on regions within the weight space where E is "flat" in the sense of small $\|\partial E\|$, enhancements are possible. A very simple strategy is to implement an inertia by assuming ∂E to be the acceleration of an object moving with friction on the error surface. The effective weight update is the gradient added by a fraction of the previous effective update; the fraction is the

"momentum term" $0 \leq \beta < 1$ and $1 - \beta$ represents the coefficient of friction in the physical model:

$$\begin{aligned} \Delta^{(i)} &:= \partial E|_{W^{(i)}} + \beta \Delta^{(i-1)}, \\ W^{(i+1)} &:= W^{(i)} - \alpha\ \Delta^{(i)}. \end{aligned}$$

This results in an asymptotic effective learning rate of $\tilde{\alpha} = \alpha/(1-\beta)$ for the update from $W^{(i)}$ to $W^{(i+n)}$ if $\partial E|_{W^{(j)}} = 0$ for $j = i, \ldots, i+n$ and growing n.

Plain gradient descent with or without a momentum term can be applied "online" in the sense that the weights are updated via $\partial E(m)$ after one pattern m or, if parts of the gradient are already known (as with RTRL), even during the processing of a pattern. Though not the precise gradient $\partial E|_W$ is computed this way, online training can successfully lead to results due to E being continuously differentiable, if the learning rate is small enough.

3.3.2 RPROP

Because of the bad practical performance of plain gradient descent, numerous enhancements have been introduced. A famous and effective one is Resilient Propagation (RPROP) as introduced in [RB92] or [Zel94]. It implements a dynamic step-size adaptation for each individual weight that is influenced only by the gradient's sign and not by its numerical magnitude. It implements a (pointwise) backtracking scheme to revert the last weight update if the error has increased. The update for one single weight ranges within ample limits like $\eta_{\min} = 10^{-6}$ and $\eta_{\max} = 50$ which are applied for numerical reasons only. The individual update is reduced by a factor η^- or enlarged by η^+ with $0 < \eta^- < 1 < \eta^+$, depending on whether the gradient sign has changed by the previous update or not. One requirement that is always accounted for but never explicitly mentioned as such in the conducted experiments is $\eta^- \cdot \eta^+ < 1$. It serves general stability so that adverse sign changes cannot result in exponential growth of the effective weight update. Let $u \in \mathbb{R}$ be a recurrent or recursive weight and i the iteration step. The weight update rule then is

$$\Delta_u^{(i)} = \begin{cases} -\eta_u^{(i)} & \text{if } \partial E/\partial u^{(i)} > 0, \\ +\eta_u^{(i)} & \text{if } \partial E/\partial u^{(i)} < 0, \\ 0 & \text{if } \partial E/\partial u^{(i)} = 0, \end{cases}$$

with pointwise positive step-matrix η that is initialised to $\eta_u^{(0)} = \eta_0 \forall u \in W$ with a fixed $0 < \eta_0 \in \mathbb{R}$ and evolves according to the following formula:

$$\eta_u^{(i)} = \begin{cases} \min\{\eta^+ \eta_u^{(i-1)}, \eta_{\max}\} & \text{if } (\partial E/\partial u^{(i-1)})(\partial E/\partial u^{(i)}) > 0, \\ \max\{\eta^- \eta_u^{(i-1)}, \eta_{\min}\} & \text{if } (\partial E/\partial u^{(i-1)})(\partial E/\partial u^{(i)}) < 0, \\ \eta_u^{(i-1)} & \text{if } (\partial E/\partial u^{(i-1)})(\partial E/\partial u^{(i)}) = 0. \end{cases}$$

The original RPROP in [RB92] proposes also a "weight-backtracking" scheme that reverts the previous weight update $\Delta_u^{(i)} = -\Delta_u^{(i-1)}$ for the case $\partial E/\partial u^{(i-1)} \partial E/\partial u^{(i)} < 0$. Note that also variants without backtracking exists, several variants and enhancements are discussed in [IH00]. The algorithm described by the formulas above (without weight-backtracking) is denoted as RPROP$^-$.

RPROP is not suited for online learning but for the so-called "batch learning" over the whole pattern set M.

3.4 Motivation and methods used here

The gradient computation method used depends on the network architecture. For LSTM (compare chapter 5), the special gradient computation method provided therein has been used. For an Elman net architecture (4.4) the gradient computation basing on RTRL has been used.

The computed gradient is normalised regarding the amount of pattern $|M|$ that are part of this gradient by dividing the gradient by $|M|$. This affects the weight update rule (see below) only in the beginning of the training and thereby ensures a slow and local start into gradient descent. It also allows to use a "balanced gradient" as described in 8.4. The gradient thereby reflects the following error function:

$$E = \frac{1}{|M|} \sum_{m \in M} E(m).$$

The weight update rule for conducted experiments was either gradient descent with momentum term or RPROP$^-$. The first has been applied for tasks that required online learning. If not stated otherwise, a learning rate of $\alpha = 0.1$ and a momentum of 0.75 has been used. For batch learning, RPROP$^-$ has been used with a slight modification: the pointwise absolute value of the first gradient was used as initialisation for the step matrix η so that the first effective weight update equals the one from plain gradient descent with learning rate $\alpha = \eta^{(0)}$:

$$\eta_u^{(0)} := \eta^{(0)} |\partial E/\partial u^{(0)}|.$$

This initialisation is referred to as "RPROP gradient initialisation". It has been used mainly for two reasons:

i) The training procedure should be able to be interrupted and resumed at the same point $W^{(i)}$ without $W^{(i+1)}$ jumping out oft a previously found solution. Using gradient initialisation, the training starts with a local behaviour which is good for testing and stability.

ii) The first step matrix is almost unrelated to the pattern set. This was observed to constitute a second, concealed layer of randomness that should be avoided in order to have the conducted experiments as traceable as possible. As each experiment is conducted several times with different random $W^{(0)}$, the original RPROP initialisation would result in another almost random $W^{(1)}$ with pointwise $u^{(1)} = u^{(0)} \pm \eta^{(0)}$ even when $\partial E/\partial u \approx 0$. But if there is no causal relation between $W^{(1)}$ and the pattern set M, $W^{(1)}$ could have been used as $W^{(0)}$ in the first place. This argumentation can be repeated until the step matrix approximates the gradient for the first time.

The impact of this method is described for one experiment in chapter 8.4.2. If not stated otherwise, the used parameters were $\eta^{(0)} = 0.1$, $\eta^- = 0.5$, $\eta^+ = 1.2$, $\eta_{\min} = 10^{-6}$ and $1 \leq \eta_{\max} \leq 10$.

The weight-backtracking of the original RPROP has been omitted because it is rather complicated. Because of this it is error prone to implement and the effort to implement does not relate well in comparison to the few impact it has on the performance according to [IH00]. Additionally, as it is only reverting single weights and not the whole update matrix, it is considered to break with the local approximation of E via ∂E: if the error increased, the whole weight update should be reverted and redone after reducing the step-size η_u for *every* weight.

As violating the strict gradient descent can be understood as the analogue to lateral thinking that actually enables the learning methods to find solutions also for hard problems, the RPROP approach however is still perfectly legitimate.

For this thesis, experiments with two variations of RPROP have been performed which are not reported in more detail:
Whenever the total error E has increased,

- the last weight update has been completely reverted and each η_u has been reduced;

- the step matrix η has been reset to the gradient.

Both methods turned out to result in a very slow training process that did not seem to converge most of the time due to many resets. This seems to be related to a coarse error surface that might be the result of the recursive net when it is processing huge input structures. Such input structures were a basis of this thesis. A training process that is stuck in one of many local minima could of course be aborted and restarted with randomised weights. This however would require a general criterion on when to abort the training. As optimising the weight update rules was not the aim of this thesis, these problems have no further been dealt with and the very easy to implement RPROP$^-$ has been used.

4 Examples of feedforward, recurrent and recursive nets

In this chapter it will be shown that the formal notation of feedforward, recurrent and recursive neural nets conforms to another that is used within the literature. Then a convention on the graphical display of nets used in this thesis will be described and a brief overview of neural net architectures is given to show different kinds of neural nets that are used within the literature to achieve different tasks. Parts of this overview are also a preliminary for the introduction of the concept of restructuring.

4.1 Transformation of recursive nets

Within the literature one common way to describe a recursive net is to define the according net function by two feedforward nets F and G, as for example found in [Ham99]. The first feedforward net maps the state of the second to the output state, which represents the output values. The second feedforward net receives input from a label and k copies of its own context neurons. Using $g = \mathcal{F}_G : \mathbb{R}^{l+km} \to \mathbb{R}^m$ a recursive function $\tilde{g}_\xi : \left(\mathbb{R}^l\right)^*_k \to \mathbb{R}^m$ is defined by

$$\begin{aligned} \tilde{g}_\xi(\perp) &:= \xi, \\ \tilde{g}_\xi(t(u_1, \ldots, u_k)) &:= g(t; \tilde{g}_\xi(u_1); \ldots; \tilde{g}_\xi(u_k)). \end{aligned}$$

$\tilde{g}_\xi$ is the *induced* (recursive) function of the function g. Eventually, using $f = \mathcal{F}_F : \mathbb{R}^m \to \mathbb{R}^{|O|}$, the net function is defined as $f \circ \tilde{g}_\xi$ where $\xi \in \mathbb{R}^m$ is the initial context. The induced recursive function of $\mathcal{F}_G$ and the output net F *define* the recursive net. This definition has the convenience that no mixup of different relations ($\to$, $\succcurlyeq$) occurs because the neural nets are reduced to their mathematical behaviour.

Without constraints on the structure of a general recursive net it is not possible to specify the appropriate feedforward nets F and G. This is due to output neurons being able to not functionally depend solely on the context neurons, but directly on

input neurons for example (as with "shortcut connections"), but F is supposed to map only from the context neurons.

Since, however, the dynamic behavior of the net results from its recursive definition, a feedforward net G, whose net function can be used to define the aforementioned recursive function, is of more interest and it can be easily defined using the following

Definition 11 (induced transition net, transition function)**.**
Let $\mathcal{N} = \left(\{1,\dots,n\}, \{1,\dots,l\}, O, F, (V,W), \to, \twoheadrightarrow, \overset{\pi}{\to}, \xi\right)$ be a recursive net having $V \in \mathbb{R}^{n\times n}$, $W \in \mathbb{R}^{k\times n\times n}$ and $m := |\mathcal{K}|$. Further let

$$\delta : \mathbb{N} \to \mathbb{N}, i \mapsto \delta(i) = \begin{cases} i & \text{if } i \in I, \\ i + km & \text{else.} \end{cases}$$

be a function that can be used to shift all non-input neurons $N \setminus I$ by km positions and π be the monotonically increasing injection fulfilling $\pi(\mathcal{K}) = \{1,\dots,m\}$. Furthermore let

$$\begin{aligned}
N' &:= \{1,\dots,n' := n + km\},\\
I' &:= \{1,\dots,l' := l + km\},\\
O' &:= \delta(O),\\
F' &:= \{f'_{\delta(i)} := f_i : i \in N \setminus I\},\\
\to' &:= \{(\delta(i),\delta(j)) : i \to j\} \cup \{(l + \pi(i) + (\kappa-1)m, \delta(j)) : i \twoheadrightarrow j, 1 \le \kappa \le k\},\\
V' &:= (v'_{ij}) \in \mathbb{R}^{n'\times n'},\\
&\quad v'_{\delta i,\delta j} := v_{ij} \forall i,j \in N,\\
&\quad v'_{l+\pi(i)+(\kappa-1)m,\delta(j)} := w_{ij\kappa} \forall i \in \mathcal{K}, 1 \le \kappa \le k,\\
\overset{\pi}{\to}' &:= \{(\delta(M),\delta(j)) : M \overset{\pi}{\to} j\}.
\end{aligned}$$

Then the feedforward net $\tilde{\mathcal{N}} = (N', I', O', F', V', \to', \overset{\pi}{\to}')$ is called the *induced transition net of* $\mathcal{N}$. This definition is well-defined because new connections are only introduced outbound from new input neurons. The net function $\mathcal{F}_{\tilde{\mathcal{N}}}$ of the induced transition net of a recursive net $\mathcal{N}$ is called the *transition function of* $\mathcal{N}$ and reckoned as a mapping $\mathbb{R}^l \times \mathbb{R}^m \times \ldots \times \mathbb{R}^m = \mathbb{R}^{l+km} \to \mathbb{R}^m$.

This allows an equivalent view on the above mentioned recursive function based on the net function of a feedforward net on the one hand, and the net function of a general recursive net on the other hand, by means of the following

Theorem 1. *Let $\mathcal{N}$ be a general recursive net with context neurons $\mathcal{K}$ and initial context ξ and $\tilde{\mathcal{N}}$ its induced transition net with net function $g = \mathcal{F}_{\tilde{\mathcal{N}}}$. Define*

$\tilde{\xi} := (\xi(i))_{i\in\mathcal{K}} \in \mathbb{R}^m$. *Then for every tree T, the state of $\mathcal{N}$ within T is equal to the output of the recursive function $\tilde{g}_{\tilde{\xi}}$ defined over the induced transition net of $\mathcal{N}$:*

$$\forall T \in \left(\mathbb{R}^l\right)_k^* : \qquad \tilde{g}_{\tilde{\xi}}(T) = (o_i(T))_{i\in\mathcal{K}}\,.$$

Proof. The proof is mere technical. It is

$$\tilde{g}_{\tilde{\xi}}(\bot) = \tilde{\xi} = (\xi(i))_{i\in\mathcal{K}} = (o_i(\bot))_{i\in\mathcal{K}}\,. \tag{4.1}$$

The net function of $\tilde{\mathcal{N}} = (N', I', O', F', V', \to', \overset{\pi}{\to}')$ is defined over the net function of $\mathcal{M} := (N', I', O', F', (V', \vec{0}), \to', \emptyset, \overset{\pi}{\to}', \vec{0})$, so let $\tilde{o}_i$ be the activation of neurons in $\mathcal{M}$. The statement will first be shown for leaves and trees with depth at most one.

Let $T = x(\bot, \ldots, \bot)$. Then it is

$$\tilde{g}_{\tilde{\xi}}(T) = g(x; \tilde{g}_{\tilde{\xi}}(\bot); \ldots; \tilde{g}_{\tilde{\xi}}(\bot)) = g(x; \tilde{\xi}; \ldots; \tilde{\xi}) = \mathcal{F}_{\tilde{\mathcal{N}}}(x; \tilde{\xi}; \ldots; \tilde{\xi}). \tag{4.2}$$

Remark that ";" denotes the vector concatenation. By definition, the input vector is formed into a leaf and issued to $\mathcal{F}_{\mathcal{M}}$. The argument is

$$U = x; \tilde{\xi}; \ldots; \tilde{\xi}(\bot, \ldots, \bot) \in \left(\mathbb{R}^{l+km}\right)_k^+$$

so that holds $\tilde{g}_{\tilde{\xi}}(T) = (\tilde{o}_i(U))_{i\in\delta\mathcal{K}}$. It will be shown that

$$\tilde{o}_{\delta j}(U) = o_j(T)\ \forall j \in N. \tag{4.3}$$

At first, remark that by definition of $\mathcal{M}$:

$$\begin{aligned}\tilde{o}_{\delta j}(U) =& f_{\delta j}\Big(\sum_{i\to'\delta j} v'_{i\delta(j)}\tilde{o}_i(U) + \sum_{(i,\delta j)\in\emptyset} \vec{0}^\top q^{-1} \circ \tilde{o}_i(U) + \sum_{J\overset{\pi}{\to}'\delta j}\prod_{i\in J}\tilde{o}_i(U)\Big)\\ =& f_j\Big(\sum_{J\overset{\pi}{\to}'\delta j}\prod_{i\in J}\tilde{o}_i(U) + \sum_{i\to'\delta j} v'_{i\delta(j)}\tilde{o}_i(U)\Big).\end{aligned}$$

Furthermore, if $J \overset{\pi}{\to}' \delta j$ holds, then $\exists! J' :\ J' \overset{\pi}{\to} j$, so $\overset{\pi}{\to}$ can replace $\overset{\pi}{\to}'$:

$$\sum_{J\overset{\pi}{\to}'\delta j}\prod_{i\in J}\tilde{o}_i(U) = \sum_{J'\overset{\pi}{\to} j}\prod_{i\in\delta J'}\tilde{o}_i(U) = \sum_{J'\overset{\pi}{\to} j}\prod_{i\in J'}\tilde{o}_{\delta i}(U). \tag{4.4}$$

Now let $K_\kappa := \{l + i + (\kappa-1)m : 1 \le i \le m\} \subseteq I'$ (i.e. the set of input neurons where the output of the context neurons of the κ-th child will be placed) and $K := \bigcup_{1\le\kappa\le k} K_\kappa$. Then δN and K_κ are pairwise disjoint and $N' = \delta N \cup K$. The

argumentation on $\overset{\pi}{\to}'$ and $\overset{\pi}{\to}$ analogously holds for $\to'$ and $\to$ if involved neurons are not within K. Note that $\to'_K = \emptyset$ since $K \subseteq I'$.

$$\begin{aligned}(4.5)\quad \sum_{i\to'\delta j} v'_{i\delta(j)}\tilde{o}_i(U) &= \left(\sum_{i\to'\delta j,\ i\in\delta N} + \sum_{i\to'\delta j,\ i\in K}\right) v'_{i\delta(j)}\tilde{o}_i(U)\\ &= \sum_{i'\to j} \overbrace{v'_{\delta(i')\delta(j)}}^{=v_{i'j}} \tilde{o}_{\delta i'}(U) + \sum_{i\to'\delta j,\ i\in K} v'_{i\delta(j)}\tilde{o}_i(U)\\ &= \sum_{i\to j} v_{ij}\tilde{o}_{\delta i}(U) + \sum_{i\to'\delta j,\ i\in K} v'_{i\delta(j)}\tilde{o}_i(U).\end{aligned}$$

By definition it is

$$\sum_{i\to'\delta j,\ i\in K} = \sum_{1\le\kappa\le k}\ \sum_{l+\pi(i)+(\kappa-1)m\to'\delta j,\ i\in\mathcal{K}}.$$

Because $\pi : \mathcal{K} \to \{1,\dots,m\}$ and π^{-1} are monotonic, it is $(\xi(\pi^{-1}(i)))_{i=1}^m = (\xi(i))_{i\in\mathcal{K}}$ and for this particular U it is $\tilde{o}_{l+\pi(i)+(\kappa-1)m}(U) = \xi(i)$, yielding

$$\begin{aligned}(4.6)\quad \sum_{i\to'\delta j,\ i\in K} v'_{i\delta(j)}\tilde{o}_i(U) &= \sum_{1\le\kappa\le k}\ \sum_{l+\pi(i)+(\kappa-1)m\to'\delta j,\ i\in\mathcal{K}} v'_{l+\pi(i)+(\kappa-1)m,\delta(j)}\xi(i)\\ &= \sum_{1\le\kappa\le k}\ \sum_{i\overset{\kappa}{\rightsquigarrow}j} w_{ij\kappa}\xi(i).\end{aligned}$$

Now using (4.6) within (4.5) and the result together with (4.4) within (4.3) and renaming bound variables it follows

$$(*)\qquad \tilde{o}_{\delta j}(U) = f_j\left(\sum_{i\to j} v_{ij}\tilde{o}_{\delta i}(U) + \sum_{1\le\kappa\le k}\ \sum_{i\overset{\kappa}{\rightsquigarrow}j} w_{ij\kappa}\xi(i) + \sum_{J\overset{\pi}{\to}j}\ \prod_{i\in J}\tilde{o}_{\delta i}(U)\right).$$

Beside the occurrence of $\tilde{}$ and δ, this equals $o_j(x())$ because the original relations are used. No reference is made to any o_i with $i \in K = N' \setminus \delta N$ because ξ is used instead. (4.3) is now shown by induction over the feedforward structure of N'. Let $\rightsquigarrow$ be the helper relation of the original recursive net N as in definition 8, $M_0 \subseteq N$ be the set of all neurons without incoming connections, $M_0 = \{n \in N : \rightsquigarrow_n = \emptyset\}$, and let $M_{i+1} := {}_{M_i}{\rightsquigarrow}$ for $i \in \mathbb{N}_0$. Then, for some $d \in \mathbb{N}_0$ it is $N = M_d$.

i) (4.3) holds for $j \in M_0$, using (2.1) it is either input or contains a sum over initial values:

$$\tilde{o}_{\delta j}(U) = \begin{cases}(x)_j & \text{if } \delta j = j \in I,\\ f_j(\sum_{1\le\kappa\le k}\sum_{i\overset{\kappa}{\rightsquigarrow}j} w_{ij\kappa}\xi(i)) & \text{if } j \in M_0 \setminus I,\end{cases} \quad = o_j(x()) = o_j(T).$$

ii) Let (4.3) be valid for all $M_i,\ 0 \leq i < e$. For $j \in M_e$, every expression $\tilde{o}_{\delta i}$ on the right hand side of $(*)$ refers to an activation of δi with $i \in M_f$ for some $f < e$ for which (4.3) holds, yielding $\tilde{o}_{\delta j}(U) = o_j(T)\ \forall j \in M_e$, that means (4.3) is valid on M_e.

iii) Finally (4.3) holds for every $j \in N = M_d = M_e$ for some $e \in \mathbb{N}_0$.

For $T = x()$ and $T = \perp$, (4.3) is valid. For $T = x(u_1, \ldots, u_k)$, $u_\kappa = \perp$ or $u_\kappa = y_\kappa()$ it is $\tilde{g}_{\tilde{\xi}}(T) = g(x, \tilde{g}_{\tilde{\xi}}(u_1), \ldots, \tilde{g}_{\tilde{\xi}}(u_k)) = \mathcal{F}_{\mathcal{M}}(U)$ with $U = x; \tilde{g}_{\tilde{\xi}}(u_1); \ldots; \tilde{g}_{\tilde{\xi}}(u_k)()$ and so the same argumentation as above holds after modifying (4.6): for every leaf $u_\kappa \neq \perp$ write

$$\tilde{o}_{l+\pi(i)+(\kappa-1)m}(U) = \left(\tilde{g}_{\tilde{\xi}}(u_\kappa)\right)_i$$

instead of $\xi(i)$ as per definition of U. Now (4.3) holds for trees of depth at most one. Since for every tree $t(u_1, \ldots, u_k)$ of depth at most D, u_κ has depth at most $D - 1$ or is $\perp$, the statement follows by induction. □

Now assume a recurrent net G to conform with the following restrictions:

i) for each input neuron i there exists one "copy neuron" j, that is, $\rightarrow_j = \{i\}$, $\succ_j = \overset{\pi}{\rightarrow}_j = \emptyset$ and $f_j = id$,

ii) there is one dummy neuron j' with $\succ_{j'} = N \setminus I$.

Then for the induced transition net holds $O' = \mathcal{K} = N \setminus I$, hence the values of a label can be accessed by connection to the copy neurons and every neuron activation is part of the context. So a feedforward net $F(G)$ can be constructed such that by theorem 1 the following identity between functions hold:

$$\mathcal{F}_G \ \equiv\ \mathcal{F}_{F(G)} \circ (\mathcal{F}_{\tilde{G}})^{\sim}_{\tilde{\xi}}\,.$$

Note that $F(G)$ and $\tilde{G} = \tilde{G}(G)$ are feedforward nets. These results apply analogously for biased networks.

Learnability

Questions about the learnability of recursive nets, that is, the expressiveness in terms of the Vapnik Chervonenkis dimension (VC dimension) and others, have been dealt with in [Ham99]. These examinations base, amongst others, on the potential to assume any tree with limited depth D to be a regular tree of depth exactly D. This is necessary because the recursive net can then be unfolded into a feedforward net of a size that is the same for any tree with a depth that is *limited* by D. In order to allow appropriate modifications of an input tree it is necessary for the net to have

special initial context and bias given. The following theorems are adapted versions of the lemmas from [Ham99, chapter 4.3] to show that for the general recursive net defined here these assumptions can also be fulfilled:

The initial context ξ is almost arbitrarily interchangeable when an additional input neuron is added to the network. For this, the depth and input dimension of the input structure must be increased by one by applying a mapping

$$\begin{aligned}\hat{}:&(\mathbb{R}^m)_k^* \to (\mathbb{R}^{m+1})_k^+,\\ &\perp \mapsto \hat{\perp} := (0,\ldots,0,1)(\perp,\ldots,\perp),\\ &\alpha(a_1,\ldots,a_k) \mapsto \alpha;0(\hat{a_1},\ldots,\hat{a_k}) \text{ recursively.}\end{aligned}$$

Remember that it is $\Sigma_k^* = \Sigma_k^+ \cup \{\perp\}$. $\hat{\perp}$ is defined as a tree (leaf) and not as a symbol for empty positions.

Theorem 2. *For every (biased) recursive net $\mathcal{N}$ with activation functions $F = \{f_i\}$, context neurons $\mathcal{K}$, initial context ξ with $\xi(i) \in f_i(\mathbb{R})$ and for every function $\xi' : \mathcal{K} \to \mathbb{R}$ there exists a (biased) recursive net $\mathcal{N}'$ with initial context ξ' so that for all $t \in (\mathbb{R}^m)_k^+$ holds: $\mathcal{F}_{\mathcal{N}}(t) = \mathcal{F}_{\mathcal{N}'}(\hat{t})$.*

Proof. Define $\mathcal{N}'$ like $\mathcal{N}$, but introduce one additional input neuron j connected to every context neuron $l \in \mathcal{K}$. The weight has no influence within instances, where labels of the original tree are processed, because by definition of $\hat{}$ a label α becomes $\alpha;0$ so it is multiplied with 0. Let $o'_l(t)$ be the neuron output function in $\mathcal{N}'$. It is only to show that w_{jl} can be chosen such that $o'_l(\hat{\perp}) = \xi(l)$.

The weight will be defined individually as such that it annuls the net input and replaces it with the preimage of the original initial context. This is done by subtracting the individual net input of the original net that would have been calculated with the new initial context ξ':

$$w_{jl} := f_l^{-1}(\xi(l)) - \sum_{i \to l} w_{il} o'_i(\hat{\perp}) - \sum_{i \overset{k}{\nrightarrow} l} \sum_{\kappa=1}^{k} v_{il}^{(\kappa)} \xi'(i) - \sum_{I \overset{\pi}{\to} l} \prod_{i \in I} o'_i(\hat{\perp}).$$

Because context neurons can receive feedforward input from other context neurons so that within the above term $i \to l$ can be true for $i \in \mathcal{K}$, the previous definition must be applied recursively, starting at neurons without feedforward input. This results in the expected output:

$$o'_l(\hat{\perp}) = f_l(\ldots + w_{jl} \cdot 1) = f_l(\ldots + f_l^{-1}(\xi(l)) - \ldots) = \xi(l).$$

□

Because of this theorem, one can assume an arbitrary initial context if only the original initial context was within range of the activation functions.

As also shown in [Ham99], the bias of a recursive net is arbitrarily interchangeable when an additional input neuron is added to the network and the input vectors from $x \in \mathbb{R}^n$ are transformed into $x' = x; 1 \in \mathbb{R}^{n+1}$, and because of this, the bias can be assumed to be any value. For brevity, the following two theorems use $net_i(t) = net_i(W, o, t)$ as the term for the net input as in definition 9 with W as the feedforward weights and the neuron activation function being explicit parameters. So for a biased net the output is $o_i(t) = f_i(net_i(W, o, t) + \theta_i)$. Note that o is the vector of all activations, $o = (o_i(t))_{i \in N}$, so the above expression contains a recursion over the feedforward structure.

Theorem 3. *For every biased recursive net $\mathcal{N}$ with activation functions $F = \{f_i\}$ and bias $\theta = \{\theta_i\}$ and for every $\theta' = \{\theta'_i\}$ there exists a biased recursive net $\mathcal{N}'$ with bias θ' so that for all $t \in (\mathbb{R}^m)^+_k$ holds:*
$\mathcal{F}_{\mathcal{N}}(t) = \mathcal{F}_{\mathcal{N}'}(t')$ with recursively $t' = \lambda(t); 1\Big(\chi_1(t)', \ldots, \chi_k(t)'\Big) \in (\mathbb{R}^{m+1})^+_k$.

Proof. Let $o_i(t) = f_i(net_i(W, o, t) + \theta_i)$ be the activation function in $\mathcal{N}$. Assume $\mathcal{N}$ to have one additional input neuron j that is not connected to any other neuron so its values will be ignored. $\mathcal{F}_{\mathcal{N}}$ then receives input labels from $\mathbb{R}^{m+1}$ instead of $\mathbb{R}^m$. It is then $o_i(\tau; x(\ldots)) = o_i(\tau; 0(\ldots))$ for arbitrary $x \in \mathbb{R}$ and the label $\tau = \lambda(t)$ of the original tree. Define $\mathcal{N}'$ like $\mathcal{N}$ but with the additional input neuron j connected to every other non-input neuron. Let $t' = \tau; 1(\ldots)$ be the modified input tree and let the output function be $o'_i(\tau; 1(\ldots) = f_i(net'_i(W', o', \tau; 1(\ldots)) + \theta'_i)$. V remains unchanged. For the new connections $j \to i$ define weights $w_{ji} := \theta_i - \theta'_i$. The output can be written using net_i from the original net and manually applying the input from the new connection:

$$\begin{aligned} o'_i(\tau; 1(\ldots)) &= f_i(net_i(W, o', \tau; 0(\ldots)) + \overbrace{(\theta_i - \theta'_i)}^{w_{ji}} \cdot 1 + \theta'_i) \\ &= f_i(net_i(W, o', \tau; 0(\ldots)). \end{aligned}$$

The equality $o'_i = o_i$ follows by induction, starting for trees that are leaves and at neurons without incoming feedforward connections. □

Remark 8. Because the bias can be chosen arbitrarily, it can for example be defined as to annul the net input for each individual neuron when the additional input is 0. This can be done for a fixed leaf $n = \beta; 0() \in (\mathbb{R}^{m+1})^+_k$ by iteratively

defining $\theta'_i := -net_i(W, o', n)$, starting at neurons without feedforward input. If the resulting net $\mathcal{N}'$ is activated for n, the result is

$$\begin{aligned} o'_i(n) &= f_i(net_i(W, o', n) + w_{ji} \cdot 0 + \theta'_i) \\ &= f_i(net_i(W, o', n) - net_i(W, o', n)) = f_i(0). \end{aligned}$$

These neuron outputs only depend on the individual activation function and not on the leaf onto which the additional weight has been adapted to.

Theorem 4. *For every (biased) recursive net $\mathcal{N}$ with activation functions $F = \{f_i\}$ and initial context within range of the respective activation functions there exists a biased recursive net $\mathcal{N}'$ and a mapping $\hat{} : (\mathbb{R}^m)^*_k \to (\mathbb{R}^{m+1})^+_k$ such that for every tree $t \in (\mathbb{R}^m)^*_k$ with depth $< D \in \mathbb{N}$ a tree $\hat{t} \in (\mathbb{R}^{m+1})^+_k$ with all its leaves at depth exactly D exists and $\mathcal{F}_{\mathcal{N}}(t) = \mathcal{F}_{\mathcal{N}'}(\hat{t})$.*

If $f_i(0) = 0$ one can assume bias and initial context 0 and the tree can simply be expanded by labels containing zeroes. Otherwise, a leaf $n()$ would be needed such that $o_i(n()) = o_i(n(n(), \ldots, n()))$ and then the state within $n()$ could be used as initial context and a tree could be expanded to a regular tree by filling it up with this one label.

However, no restrictions according to the weights are given, so the state transition cannot be guaranteed to be a contraction and hence the Banach fixed-point theorem cannot be applied to find an initial context that is a fixed point. Also it is not guaranteed that $f_i^{-1}(0)$ exists, otherwise the bias could be specified to result in this value. But this problems can be overcome by introducing an additional neuron:

Proof. Assume $\mathcal{N}$ to have the initial context $\xi(i) = f_i(0)$. Transform it to have the annulling bias according to $n = \alpha; 0()$ as defined in remark 8 with one additional input neuron and further assume the net to have another additional, unconnected input neuron j. Let $o_i(t) = f_i(net_i(W, o, t) + \theta_i)$ be its neuron output function for a tree $t = \tau; x; y(\ldots) \in (\mathbb{R}^{m+2})^+_k$. Because of the annulling bias it is $o_i(\alpha; 0; 0()) = f_i(0) = \xi(i) = o_i(\perp)$. Define a biased recursive net $\mathcal{N}'$ like $\mathcal{N}$ but with the additional input neuron j connected to every non-input neuron. Using the original net_i let the neuron output be $o'_i(\alpha; 0; x(\ldots)) = f_i(net_i(W, o', \alpha; 0; 0(\ldots)) + w_{ji}x + \theta_i)$. The weight will be defined such that activating $n' := \alpha; 0; 1(\alpha; 0; 0(), \ldots, \alpha; 0; 0())$ yields the same state as $\alpha; 0; 0()$. This is done iteratively over the feedforward structure with $w_{ji} := -net_i(W, o', n') - \theta_i$. Note that for calculating $net_i(\ldots, \alpha; 0; 0)$ the weight w_{ji} disappears in $w_{ji} \cdot 0$ so this is not an implicit definition even though n' is a tree of depth 1. Using these weights results in the output

$$o'_i(n') = f_i(net_i(W, o', n') + w_{ji} + \theta_i) = f_i(0) = o_i(\alpha; 0; 0()).$$

When using this net, a tree t can be modified to $\lambda(t);1;0(\ldots)$ at each label, expanded with labels $\alpha;0;1$ at depth $< D$ and with label $\alpha;0;0$ for the leaves at depth $= D$. □

Figure 4.1 shows an example about how an input structure looks like after undergoing the modification needed within the above theorem.

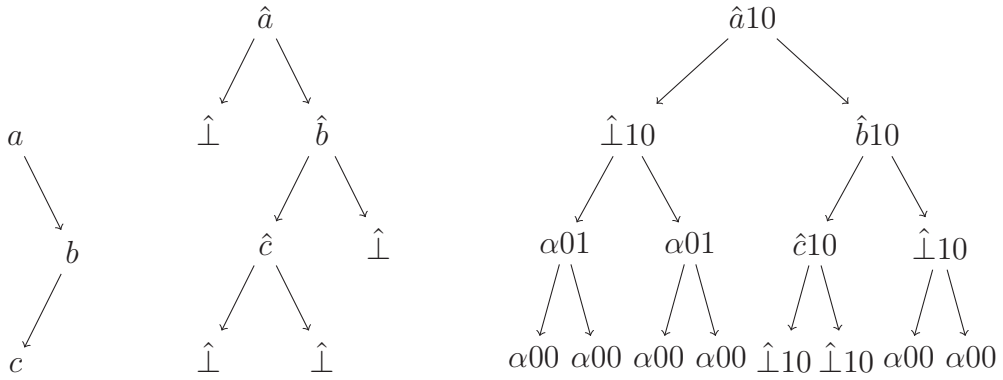

Figure 4.1. *Example for the expansion to a regular tree with depth 3. There is* 1 *additional input after transformation for the initial context (left to middle). The nodes* $a;0$ *are renamed to* $\hat{a}$ *etc. and* $\hat{\perp} = \vec{0};1()$ *is introduced. Then (middle to right) two additional inputs are added for the two new input neurons, the first for the bias which is* 1 *for labels in subtrees that are present in the middle tree and* 0 *for those added during expansion. The second input is* 1 *in every added non-leaf to create a state as if it were a leaf. As* α *can be chosen arbitrarily, it could be defined as* $\hat{\perp}$*, but this would conceal its usage in this example. The expansion could be performed to any larger depth.*

4.2 The architectural graph

Architectures for recurrent nets are often described by defining their net function and depicted using graphs. Depicting feedforward nets is rather unambiguous and can be done using directed acyclic graphs (DAGs). To represent neurons, nodes are used with optional labels containing their respective activation function (if not all have the same) and edges represent the connection between the nodes with optional labels at the edges representing fixed weights.

The graphical description of a recurrent or recursive net architecture requires to distinguish between recursive and feedforward connections. For simple architectures this is often done by creating cycles with (usual) edges pointing against the overall direction of all other edges. But with enough cycles it becomes unclear whether the connection is meant as recursive or as feedforward.

In this thesis, the architectures are described by requirements that are depicted in a graph-like figure according to the following rules:

i) Neurons are depicted as circles. A set of neurons can be depicted as one rectangle. The actual numbers corresponding to the neurons are usually withhold but can be written within the circle or rectangle as a number, range or placeholder. The size of the neuron set can be written inside or aside the rectangle with a "#" prefix. Input and output neurons and neuron sets are made explicit with the letters "I" and "O". Rectangles containing the expression q^{-1}, Q or $K_{m,n}$ have a special meaning (see below).

ii) A connection $a \to b$ (an element $(a, b) \in \to$ of the feedforward relation) is depicted as a directed edge from a inbound to b with the arrow pointing at b.

iii) Connections $a \succ b$ are depicted like $a \to b$ but starting with a bullet "$\bullet$" to indicate the underlying scalar product that is added within the net input of b due to this connection.

iv) Multiplicative connections $A \xrightarrow{\pi} b$ are considered unidirectional hyperedges between all $a \in A$ and b and they are depicted as multiple lines starting at b and each a joining in one "$\otimes$" symbol while on the line connected to b an arrowhead points at b.

v) Edges or lines ("connections") between circles and rectangles are placeholder for the set of connections to or from all circles which the rectangle represents. Between rectangles they are a placeholder for connections between each pair of circles originating from the corresponding rectangles, thereby adding the Cartesian product of the corresponding neuron sets to the relation ($\to$, $\succ$ and possibly $\xrightarrow{\pi}$).

vi) Connections can have a label stating the respective weight. The weight (weight tensor) belonging to a recursive connection can be written as a vector of weights (weight matrices) of length k.

vii) Rectangles containing the expression q^{-1} or the letter Q ("q-box") can only be connected by directed or undirected edges but not with edges starting or ending in a bullet or with lines originating from a hyperedge. An undirected edge to a q-box is a placeholder for two directed edges forming a cycle.

The q-box itself does not represent a neuron set but elements in $\succ$. Let $n_1, \ldots, n_i$ be the neurons with undirected edges to the q-box, $r_1, \ldots, r_j$ be the neurons with edges pointing *from* (reading from) it and $w_1, \ldots, w_l$ be the

neurons with edges pointing *at* (writing to) it. The q-box then represents the requirement

$$\{n_1, \ldots, n_i, w_1, \ldots, w_l\} \times \{n_1, \ldots, n_i, r_1, \ldots, r_j\} \subseteq \succcurlyeq .$$

viii) When a recursive net is unfolded into a feedforward net for a fixed input structure, the q-box will be transformed into a complete bipartite graph that is depicted as a rectangle containing the expression $K_{m,n}$ where $m = (i + l)k$ and $n = i + j$ with variables as in vii) and k being the maximum fanout of the underlying recursive net. It does not represent a neuron set but abbreviates the $m \cdot n$ directed edges (from each incoming to each outgoing neuron) into $m + n$ edges to be depicted at roughly half the length.

Examples of usage are given in the following sections where different kinds of feedforward, recurrent and recursive neural net architectures are introduced and one example of viii) can be found at page 52 in figure 5.3.

Remark 9. As mentioned in remark 4, the relations $\rightarrow$, $\succcurlyeq$ and $\xrightarrow{\pi}$ are respectively defined as the intersection of all relations fulfilling the given requirements. As a result, using two separate hidden layers which are each connected to a q-box will result in smaller relation sets. This is due to recursive connections between these layers being implicitly removed. See figure 6.2 at page 63 for an example.

4.3 Multilayered networks

A multilayer feedforward network (multilayer perceptron, *MLP*) is a biased feedforward net $(N, I, O, (F, \Theta), V, \rightarrow, \emptyset)$ with the following properties:

i) N is partitioned into $N_0, \ldots, N_{h+1}$ (*layers*) where $n_l := |N_l| \in \mathbb{N}$ for $0 \leq l \leq h + 1$ and $h \in \mathbb{N}_0$.

ii) $I = N_0$ (*input layer*), $O = N_{h+1}$ (*output layer*).

iii) The connection structure complies with $\rightarrow \subseteq \bigcup_{i=1,\ldots,h+1} N_{i-1} \times N_i$.

$N_1, \ldots, N_h$ are called the hidden layers. All neurons within the hidden layers are usually activated by using the same activation function f and the output layer is linearily activated, that is, $f_i = id,\ i \in O$. Because of the layered structure, the output of a whole layer can be expressed using vector arithmetic by

$$o_{N_l}(x) = f\left(V_l^\top o_{N_{l-1}}(x) + \Theta_l\right) \in \mathbb{R}^{n_l},\ V_l \in \mathbb{R}^{n_{l-1} \times n_l}, \Theta_l \in \mathbb{R}^{n_l}$$

where f is applied pointwise and $(V_l)_{ij}$ is the weight for the connection from the i-th neuron in N_{l-1} to the j-th neuron in N_l.

In the special case of a MLP with one hidden layer, one output neuron and linear output activation the resulting mapping function can be reduced to

$$\mathcal{F} : \mathbb{R}^l \to \mathbb{R}, \quad x \mapsto c^\top f\,(Ax + b) + \theta$$

with $A \in \mathbb{R}^{h \times l}, c \in \mathbb{R}^h, b \in \mathbb{R}^h, \theta \in \mathbb{R}$, where h denotes the size of the hidden layer. In [Hor89] it has been shown that such a MLP is capable of approximating any continuous function $f : K \to \mathbb{R}$ on a compactum $K \subset \mathbb{R}^l$ arbitrarily well in $\|.\|_\infty$ norm using one hidden layer of sufficient size (and the bias $\theta \in \mathbb{R}$ is actually not needed, that is, $\theta = 0$). This property is called *universal approximation capability*.

Recursive MLP

Within the literature, a recursive MLP is defined over the net function $h \circ \tilde{g}_\xi$ that results from two feedforward MLP with net functions h and g. Such a net function is equal to that of a general recursive net that is acquired by expanding the definition of a MLP with $h_1 + 1 + h_2$ hidden layers, (equally the composition of two MLPs with h_1 and h_2 layers respectively). The definition is expanded by introducing the recurrent connections $\succcurlyeq = M_{h_1+1} \times M_1$, that is, the first MLP's output layer serves as recursive input for its own first hidden layer and as feedforward input to the second MLP.

Figure 4.2 shows the architectural graphs according to these networks.

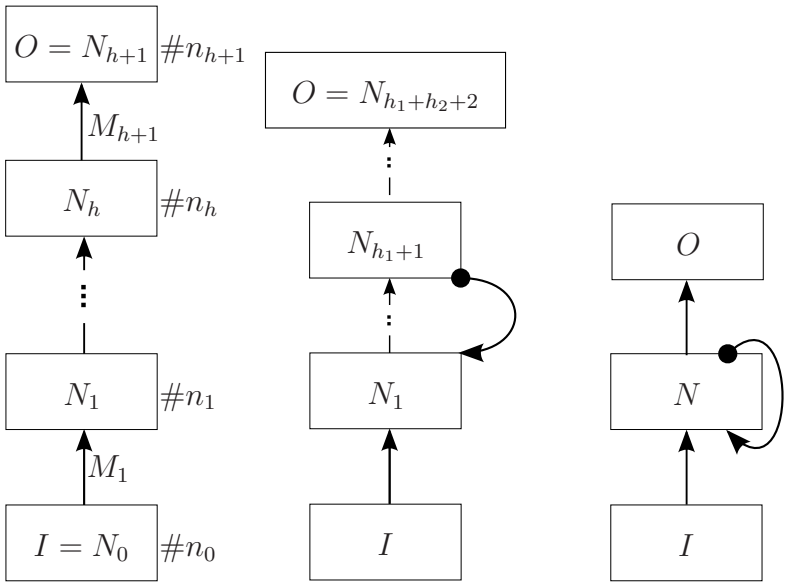

Figure 4.2. *MLP, Recursive MLP, Elman net.*

4.4 Elman nets

When two MLPs without hidden layers ($h_1 = h_2 = 0$) and output activation function f are formed into a recurrent net by adding the connections $\succcurlyeq = M_1 \times M_1$, the recurrent net is called *Elman net* or *Simple Recurrent Net.* It can be enhanced to a recursive net as well by allowing $k \geq 2$.

When the net function is to be described in vector arithmetic, the input weight matrix $V_I \in \mathbb{R}^{l \times h}$ and context weight matrices $W_\kappa \in \mathbb{R}^{h \times h},\ 1 \leq \kappa \leq k$ can be used. Let $V_O \in \mathbb{R}^{h \times |O|}$ be the weight matrix to the output layer, then for a sequence $T = (x, y, z) \in (\mathbb{R}^l)^+$ the net function is

$$T = (x, y, z) \mapsto f(V_O^\top f(V_I^\top z + W_1^\top f(V_I^\top y + W_1^\top f(V_I^\top x + W_1^\top \xi)))),$$

for a leaf $T = x() \in (\mathbb{R}^l)_2^+$ it is

$$T = x() \mapsto f(V_O^\top f(V_I^\top x + W_1^\top \xi + W_2^\top \xi))$$

and for an input tree $T = x(y(), z()) \in (\mathbb{R}^l)_2^+$ using $\xi = \vec{0}$:

$$T = x(y(), z()) \mapsto f(V_O^\top f(V_I^\top x + W_1^\top f(V_I^\top y) + W_2^\top f(V_I^\top z))).$$

Figure 4.2 contains the architectural graph of this network architecture.

Elman net emulating a recurrent MLP

As described in [Ham99, chapter 4.3] or to full detail in [Ham97], for any recurrent MLP an Elman net with the same number of weights $\neq 0$ and neurons can be constructed that emulates the recurrent MLP, if time skips of appropriate length are introduced to the input data; the state within the i-th time skip contains the correct activation of the $i + 1$-th layer. This way, dealing with a recurrent MLP on a sequence of length n can equally be seen as dealing with an Elman net on a sequence of length hn with a constant factor h. For trees, the skip-nodes must be placed at any fixed position and each final skip-node must be placed at the position the original node was placed.

4.5 Bidirectional recurrent nets

Introduced in [SP97], a bidirectional recurrent neural net (BRNN) consists of a hidden layer that is split into two parts. Each part is fully connected to itself and not connected to the other, the net operates on sequences. The recurrent connections

from one part however are not meant to access activations from previous parts, but from future parts of the sequence by means of the q^{+1} operator:

$$q^{+1}(o_i(t)) = o_i(t+1).$$

Since the two parts of the hidden layer are not connected to each other, the mapping function is well-defined. In the original paper a recurrent Elman net has been used with isolated hidden parts and an output layer receiving input from both parts. However, the concept of a bidirectional recurrent net can also be defined over different network structures. Using the originally proposed Elman net but with linear output activation, the mapping function can be expressed in vector arithmetic by

$$\begin{aligned} &f_{BRNN} : (\mathbb{R}^l)^+ \to (\mathbb{R}^m)^+, \\ &\quad x = (x_k)_1^n \mapsto \left(V_O^\top \cdot \mathcal{F}_F((x_1, \ldots, x_k)); \mathcal{F}_B((x_n, x_{n-1} \ldots, x_k))\right)_{k=1}^n \end{aligned}$$

with $V_O \in \mathbb{R}^{2h \times m}$ being the connection matrix to the output layer, $\mathcal{F}_F$ and $\mathcal{F}_B$ being the output functions of two independent Elman nets F ("forward") and B ("backward") of the same size h with output layers copying the respective hidden layer. Notice that in the above expression, the argument to $\mathcal{F}_B$ is a subsequence of the original sequence x in reverse order, starting with the last element vector x_n.

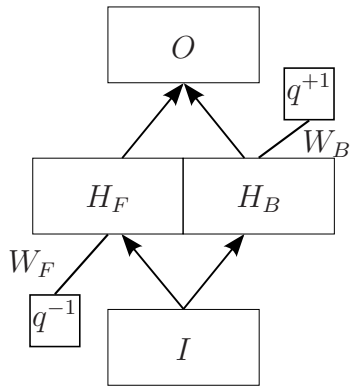

Figure 4.3. *Architectural graph of a bidirectional recurrent net. W_F and W_B are the weight matrices of the underlying Elman nets "forward" and "backward".*

The architectural graph can be seen in figure 4.3. The same net function can also be acquired by a recursive Elman net without using the q^{+1} operator when the input sequence is properly formed into a set of trees, compare chapter 6.2.2.

4.6 Echo State Networks

Echo State Networks are recurrent networks with the architecture of an Elman net with the restrictions that the weight matrices from input and hidden layer to the hidden layer are random (which holds after initialisation for gradient descend, compare chapter 3) but sparse, that is, they contain only few weights different from zero. These weights are not trained. For training, only the weights to the output layer are adapted.

These networks have similarities to the "training method" of weight guessing (compare chapter 5.3) in the sense that they can be understood as a set of randomly generated recurrent Elman nets of strongly varying size which are joined through few additionally introduced connections between the hidden layers. Thereby they are merged into one bigger hidden layer.

Vice versa an Echo State Network could be analysed for the cliques within the graph related to $\succcurlyeq$. A set of cliques that is a covering of the neurons with least interconnections possible could be understood as a set of Elman nets, each of which having the size of the respective clique that was merged through additional (inter)connections.

4.7 Cascade Correlation

Cascade Correlation (CC) is the name of several methods for incremental construction of feedforward and recurrent/recursive nets. They are motivated by the fact that it is unknown how to choose the size of, for example, a hidden layer, prior to adapting the free parameters (weights). A survey on algorithms constructing feedforward nets can be found in [Sta03]. The incremental process enlarges the number of hidden neurons within an architecture during the training process, that is, for a given task. The algorithms require certain limitations to the connection structure that differ over the actual strategies. But in general, the *result* of a CC algorithm is comparable to figure 4.4.

When ignoring the input neurons, for the weight matrices holds that the feedforward weight matrix is a triangular matrix (essentially due to the incremental construction) with zeroes at the diagonal (due to acyclic property). The recurrent weight matrix, respectively each layer of the weight tensor, is a triangular matrix. For the first recurrent CC architecture that was introduced the recurrent weight matrix was required to be a diagonal matrix, meaning each neuron was recurrently connected only to itself.

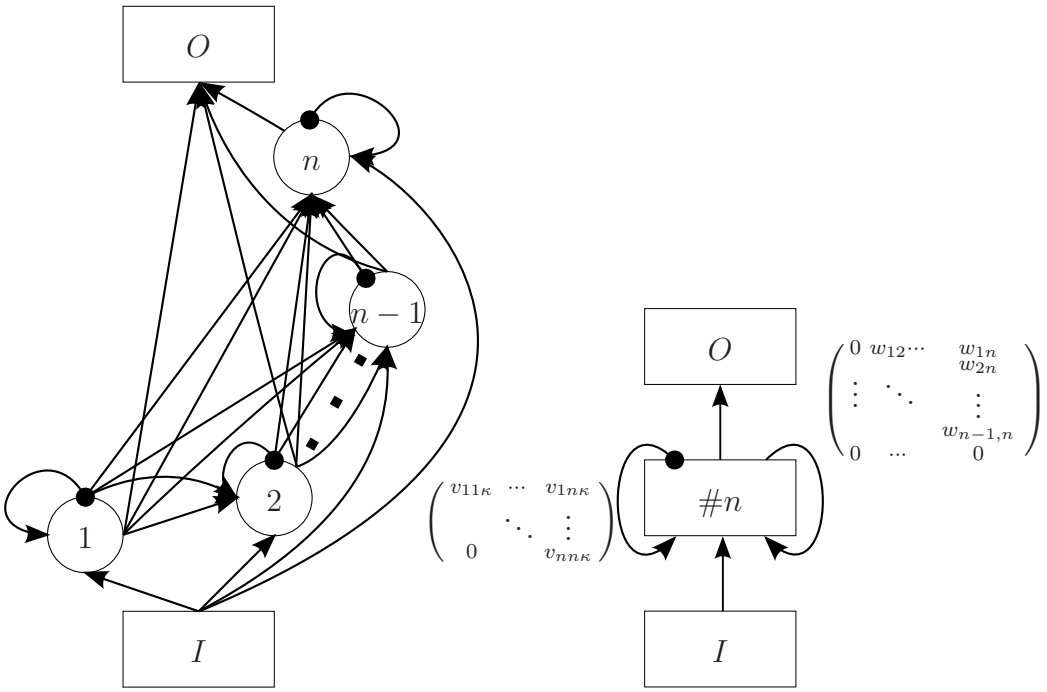

Figure 4.4. *General structure of the result of a (recursive) Cascade Correlation training process. When n neurons have been added so far, the i-th neuron has i incoming and $n+1-i$ outgoing feedforward connection as well as (up to) i incoming respectively $n-i$ outgoing recursive connections. Each neuron is connected to the output layer. All iteratively added neurons form one single hidden layer, as every added neuron is connected to all previous ones.*
Both pictures describe the same architecture: within the right one, the connection structure is hidden in the specific weight matrix and the weight tensor.
The enumeration of the hidden neurons has been chosen for ease of reading. Puristically sticking to the definition of a recursive net, it would restrict the input neuron set to be empty because the first hidden neuron is $1 \in N \setminus I$ but $I = \{1, \ldots, l\}$.

The generalisation of recurrent CC has been established in [SS97] and to contextual CC in [MSS04]. The latter makes use of the q^{+1}-operator. Using this, cyclic connection structures are to be avoided because a functional cycle would occur, making the output activation an implicit function system. This mainly results in the q^{+1}-operator being a pointer to a neuron (neurons activation). The methods described in this thesis however (chapter 6, namely Leaf Level Mirroring) do not base on neural nets but only use them as a realisation. The difference between the (connectionistical) contextual approach and the approach within this thesis can be seen as the difference between pointing to neurons versus pointing to data. In the latter case, functional cycles cannot occur.

5 Long Short-Term Memory - LSTM as a recursive net

Long short-term memory, or LSTM for short, is a general recurrent net with a novel connection structure that can be trained with a computationally inexpensive online gradient descent method. It has been introduced in [HS97] in 1997 and is reported to be appropriate to solve learning tasks containing long term dependencies of more than 1000 time steps.

LSTM has been introduced as a recurrent net. Although not intuitive due to self-recurrent neurons with a constant 1-valued weight, it can easily be introduced as a recursive net as well. This holds even though the constant 1-valued recurrent weight that creates the "constant error carrousel" ("CEC"), which is the foundation of LSTM, will be replaced with a $(1, \ldots, 1)$-vector that intuitively seems to destroy the constancy. However, the constant *error* carrousel is about propagating error signals *back* by means of BPTT and since error signals are propagated back within a certain input structure, only one component of the vector is chosen, retaining the constancy needed. The transfer of LSTM to a recursive net to process structures is out of the scope of this thesis but has been dealt with in the authors diploma thesis [Arn08].

Though LSTM has been reported to solve problems where other recurrent nets failed, the discussion of [HS97] states that problems of the kind "strong delayed XOR" could still not be solved for the possible reason that storing previously observed symbols cannot help when the correct solution is approached step-wise.

The proposed learning algorithm is a modification of RTRL that has only complexity of $\mathcal{O}(1)$ per weight. It has been designed to avoid the computation of error signals that might vanish anyhow due to exponential error decay, while error signals of linear activated, non biased CECs are scaled with factor 1 and their activations are only manipulated when the input has value different from zero.

5.1 Architecture

LSTM is a biased recursive net $\mathcal{M} := (N, I, O, (F, \theta), \mathcal{W}, \rightarrow, \succ, \overset{\pi}{\rightarrow}, \xi)$ given by certain sets and relations. The structure of LSTM can be described as a set of memory blocks, each of which consists of one input and output gate and a fixed amount of memory cells. Each memory cell consists of a cell input neuron, a CEC, a scaling neuron and a cell output neuron. The input to the CEC is acquired by multiplying the activations of the cell input neuron with the (block) input gate. The cell output is acquired by multiplying the activation of the scaling neuron and the block output gate. The scaling neuron has a usual squashing function as activation and receives input only from the CEC, which has the identity as activation function. The mathematically complete description is as follows:

- The set of neurons N is partitioned into I, L and O: $N = I \cup L \cup O$, where I and O are the input and output neurons and L is the actual hidden layer that is further partitioned as $L = H \cup G_{in} \cup C_{in} \cup C_s \cup C_h \cup G_{out} \cup C_{out}$.
  - Neurons in H are called regular neurons ($H = \emptyset$ is allowed),
  - those in G_{in} and G_{out} are called input and output gates,
  - C_{in} and C_{out} contain the cell input and cell output neurons of memory blocks,
  - neurons in C_s are called CECs and
  - those in C_h are called scaling neurons.
- It is $|G_{in}| = |G_{out}| =: B$ and B is the number of memory blocks.
- The sets $C_{in}, C_s, C_h, C_{out}$ are partitioned into $C_{in,j} = \{c^i_{in,j}\}$, $C_{s,j} = \{c^i_{s,j}\}$, $C_{h,j} = \{c^i_{h,j}\}$, $C_{out,j} = \{c^i_{out,j}\}$ respectively, with $|C_{in,j}| = |C_{s,j}| = |C_{h,j}| = |C_{out,j}| =: B_j$ for $j = 1 \ldots B$. B_j is the size of the j-th memory block, so every block can have a different size.

Neurons of the same block share the same input and output gate $g_{in,j} \in G_{in}$, $g_{out,j} \in G_{out}$ and can be summarised into memory blocks:
$S_j = \{g_{in,j}, g_{out,j}\} \cup C_{in,j} \cup C_{s,j} \cup C_{h,j} \cup C_{out,j}$. Compare figure 5.1.

The connections, that is, elements of $\rightarrow, \overset{\pi}{\rightarrow}$,and $\succ$ between neurons exist as feedforward connections on the one hand, namely between input/output layer (through shortcut connections, if used) and input/hidden layer and on the other hand they exist as recursive connections between the gates, cell input and output neurons and - if used - the regular hidden layer H. Within a block, the CECs $c_{s^v_j} \in C_{s,j}$

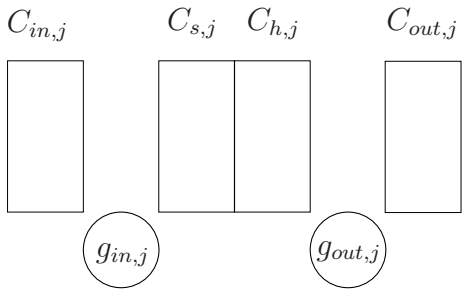

Figure 5.1. *LSTM memory block. Rectangles represent sets of neurons, circles represent single neurons. Their alignment indicates the feedforward connection structure within a memory block. All rectangles contain the same amount of neurons. Several of these figures form a set of memory blocks; together with the regular neurons from H the complete hidden layer is formed.*

have only one recursive connection to themselves and apart from that only (multiplicative) feedforward connections. This connection is fixed and is not to be trained: $v_{jj} := \vec{1} = (1, \ldots, 1)^\top \; \forall j \in C_s$. The CECs can be imagined to be embedded by feedforward connections into other neurons that mainly possess recursive connections.

The **feedforward connections** are defined as

$$\to = I \times \Big(O \cup H \cup G_{in} \cup G_{out} \cup C_{in}\Big) \cup \Big(C_{out} \cup H\Big) \times O \tag{5.1}$$

and the **recursive connections** are defined as

$$\succcurlyeq = \Big(H \cup G_{in} \cup G_{out} \cup C_{out}\Big) \times \Big(H \cup G_{in} \cup G_{out} \cup C_{in}\Big) \cup \mathrm{id}\,|_{C_s}. \tag{5.2}$$

The identity function id is written as $\{(x, f(x)) | f(x) = x\}$ and $\mathrm{id}\,|_{C_s}$ is the restriction onto $x \in C_s$. This means that CECs, that is, elements of C_s, only have one single recursive connection and each CEC only to itself. Compare figure 5.2.

Feedforward connections in $I \times O$ are sometimes called **shortcut connections**.

In $\to$ not all necessary connections are represented:

$$\begin{aligned} \xrightarrow{\pi} = \bigcup_{j=1}^{B} \bigcup_{i=1}^{B_j} \Big\{ & \Big(\{c^i_{in,j},\ g_{in,j}\}, c^i_{s,j}\Big), && \text{(input into CEC)} \\ & \Big(\{c^i_{s,j}\}, c^i_{h,j}\Big), && \text{(scaling neuron)} \\ & \Big(\{c^i_{h,j},\ g_{out,j}\}, c^i_{out,j}\Big)\Big\} && \text{(output of memory cell).} \end{aligned} \tag{5.3}$$

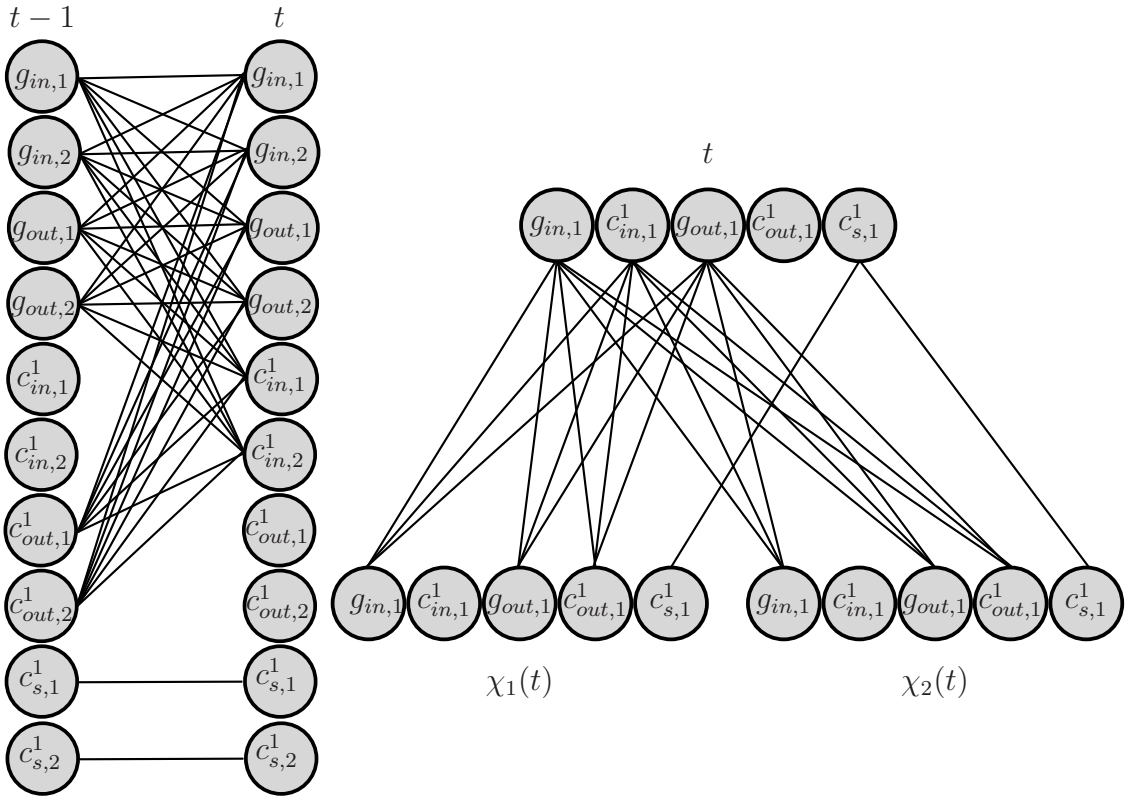

Figure 5.2. *Unfolded recursive connections for a LSTM net without regular hidden neurons. Left: two blocks of size one for sequence input ($k = 1$); nodes of same conceptual kind are arranged together in order for showing that they have in- or outgoing connections. Right: one block of size one for fanout 2 (i.e. binary tree input).*

The gates act as "gates" by being part of a *multiplicative connection* from the source to the destination. All elements of $\xrightarrow{\pi}$ do not, by definition of the net function, represent a free parameter (weight) to be trained. The relation $\xrightarrow{\pi}$ connects the neurons as seen in figure 5.1 from left to right while multiplying with the activation of the block gates. Though scaling neurons only have singletons as input sets and therefore could be defined within $\rightarrow$ as well, they are defined to have a fixed weight 1. For each neuron, special activation functions are described in [HS97]:

$$f_k(x) = \begin{cases} \operatorname{sgd}(x) & \forall k \in G_{in}, G_{out}, H, O, \\ 4\operatorname{sgd}(x) - 2 & \forall k \in C_{in}, \\ 2\operatorname{sgd}(x) - 1 & \forall k \in C_h, \\ x & \forall j \in C_s, C_{out}. \end{cases} \tag{5.4}$$

The scaled and shifted sigmoidal function contains 0 with an open neighbourhood while the sigmoidal function itself has 0 only as a limit[1]. The connections described in (5.1), (5.2), and (5.3) completely describe $\rightarrow$, $\overset{\pi}{\rightarrow}$ and $\succ\!\!\succ$. **Biases** for CECs, scaling neurons and cell output neurons are fixed to 0, other biases can be chosen arbitrarily. It was reported that certain values (i.e. a rather huge negative bias for input gates) can speed up the learning process (see next chapter).

It should be noted that neurons $k \in O$, H, G_{in}, G_{out} or C_{in} have activations without elements of multiplicative inputs and therefore look quite usual:

$$net_k(t) = \sum_{l \rightarrow k} w_{lk} o_k(t) + \sum_{l \succ\!\!\succ k} v_{lk} \bullet q^{-1} o_k(t).$$

Only neurons within C_s, C_h and C_{out} have different activations, each of which does not contain free parameters:

$$o_{c^i_{s,j}}(t) = \vec{1} \bullet q^{-1} o_{c^i_{s,j}}(t) + o_{c^i_{in,j}}(t) o_{g_{s,j}}(t), \tag{5.5}$$

$$o_{c^i_{h,j}}(t) = f_{c^i_{h,j}}\left(o_{c^i_{s,j}}(t)\right), \tag{5.6}$$

$$o_{c^i_{out,j}}(t) = o_{c^i_{h,j}}(t) o_{g_{out_{s,j}}}(t). \tag{5.7}$$

In figure 5.3 an LSTM net without regular neurons and output layer of size 1 operating on a sequence of length 3 is displayed by unfolding into a feedforward net.

Cell input neurons, gates and memory cells are not connected to the output neurons, but only to the cell output. Therefore one important observation should be made regarding the recursive connections (i.e. the way an LSTM net interacts with previous instances of itself): within $\succ\!\!\succ$ the neurons from C_{out} occur only on the left side while those from C_{in} occur only on the right. In other words: cell output neurons never receive immediate input from children (respectively previous time steps) and cell input neurons never yield immediate input to parents.

Basing on these definitions the network structure looks very complicated in comparison to a MLP or an Elman net. However, because of the fact mentioned before and because one could imagine to deal with memory blocks of size 1 only (as different input or output gates could have equivalent activation, rendering the net only computationally inefficient), a memory block of size 1 can be imagined as *one big*

[1] Because of this fact and because one could assume that a gate should be able to reach "true" 1- and 0-values it could also make sense to swap the functions between G_{in} and C_{in}, as well as between G_{out} and C_h. However, these are the activations functions used in the original paper.

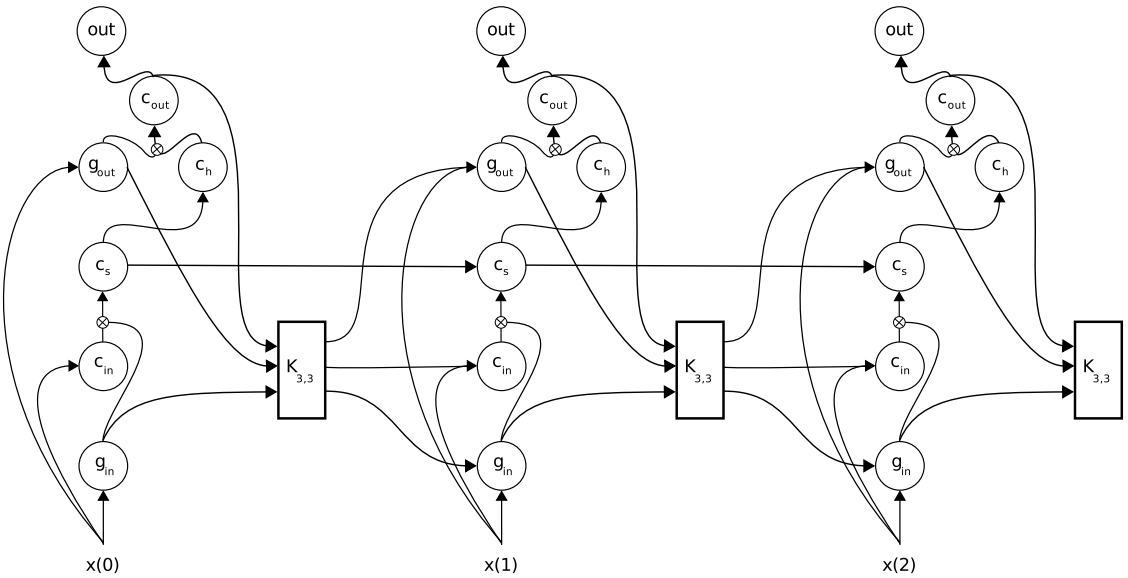

Figure 5.3. *Unfolded recursive LSTM net (one block of size 1) for an input $x \in \mathbb{R}_1^+$ of length 3. $K_{3,3}$ is a complete bipartite graph, compare viii) in chapter 4.2.*

neuron that has three times as much incoming connections as a regular neuron (two for the gates one for the cell input). Figure 5.4 contains an example for such an illustration on the left. On the right, an illustration of the "big neurons" is given.

The (multiplicative) feedforward connections divide a memory cell into one part that collects data from the context perceived by the recursive connections ($n \in C_{in}$) and another part, that exhibits their content as (a part of) the context of parent nodes.

Any general recursive net can be unfolded into a feedforward net for input structures of fixed size by appropriately copying the weights. So this can also be done with LSTM. Examples for the recursive connections are shown in figure 5.2. Most notable are the single connections between CECs ($n = c^i_{s,j}$), the missing connections from cell input to cell output and the complete connections between the rest.

Principle of operation

The way LSTM works is easily seen when operating on a sequence. The in- and output gates possess the sigmoidal function as activation function, its values range within the limits from 0 to 1. If the net input of a gate is for example < -4, then its activation is almost 0. For a rather huge positive net input, the activation is almost 1. The activation of a cell input neuron $o_{c_{in}}$ will be multiplied with those of the input gate, meaning that a gate zeroed at time steps $t - q, \ldots, t$ will prevent (strong) modification of the memory cells of the whole block.

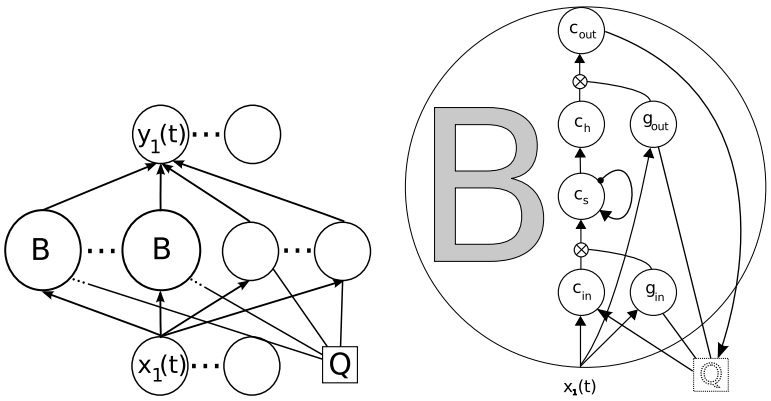

Figure 5.4. *Shortest depiction of LSTM. Left: the blocks are displayed just as enlarged neurons marked with"B". Oriented edges denote feedforward connections. Edges from the input layer and to the output layer are only drawn for one neuron each, no shortcut connections. To be fully describing, the connections to the q-box are needed, compare vii) in chapter 4.2. Without LSTM blocks "B", this is just an Elman net.*

For the content of each CEC then holds:

$$o_{c_s}(t-q) \approx o_{c_s}(t-q+1) \approx \ldots \approx o_{c_s}(t).$$

The content of the CEC will be scaled with factor 1 (the constant self-recurrent weight) and added with (almost) 0. The result will be activated by $f(x) = x$. This keeps c_s almost unchanged. However, the output range of c_s is now unrestricted which is circumvented by using the scaled output c_h. The memory cell can store information (and acts as a CEC during error backpropagation) but by the scaling neuron the block still produces restricted output.

Input gate and cell input neurons form two units that can independently control *which* information (i.e. net input) is calculated and *if*, or better *to what grade*, this information manipulates the memory.

The output gate can control, when the scaled output is actually having impact as part of the context. It has been reported that LSTM nets without output gates (but direct connections to the scaling neuron) have also worked well.

Enhancements

LSTM has been subject to modification and improvement. For example another gate has been introduced in [GSC00] that is attached to the self-connection of the CEC for being able to directly manipulate its content ("forget gate"). This capability has there shown to be useful for time-series prediction because huge

values of memory cells can be reset within one time step to zero in opposite to taking up to as much iterations as it has taken to reach this value. For example, having cell input ≈ 2 and input gate ≈ 1 for n time steps within a sequence yields memory cell value $\approx 2n$. If the learning task requires this cell to have memory cell value 0 at the $n+1$-th time step, another memory block would be needed to suite this task, or it would take additional $2n$ steps to reach the cell value 0 again - the cell would not be of use in the meantime.

Additionally, "peephole connections" have been introduced in [GSS02] being (feed-forward) connections *outgoing* directly from the memory cells to the gates of the according block. They have shown to be useful for learning tasks requiring precise timings, for example to distinguish between time series containing sharp spikes with pauses of slightly different length.

This argumentation deals with the dynamic behaviour of the inner states during activation, the enhancements can be understood to increase the flexibility of the state transition function.

"Multidimensional LSTM" was introduced in [GFS07] and is defined on k-dimensional input data. It is an actual instance of a "truly" recursive LSTM (i.e. $k \geq 2$) that is, however, no actual enhancement of LSTM itself.

5.2 Training with gradient descent

LSTM has been introduced together with an efficient combination of RTRL and BPTT in [HS97] and the application of what is called "truncation" throughout the relating papers. As a general recursive net, LSTM can of course also be trained by plain gradient descent, for example with conventional BPTT (limiting the step wise adaptation of the weights to offline, that is, once per sequence) or conventional RTRL (increasing the computational complexity *per weight* from $\mathcal{O}(1)$ to $\mathcal{O}(W)$ having W the total number of weights). In the following, the main properties of the algorithm are described.

Initialisation of biases

Some problematic behaviour of memory blocks has been discovered in [HS97, p.8, "Abuse problem and solutions"]: first, several memory blocks could tend to store the same contents. Second, blocks could tend to produce constant output, having them being "abused" as additional bias neurons and taking long time of training for them to desist from this behaviour. Finally, memory cell contents could drift off

very fast because of an input gate with rather huge positive values which implies derivatives of the scaling neurons to be almost 0.

For these problems, a simple solution was described: the gate to the blocks are biased with $-1, -2, \ldots$ or $-2, -4, \ldots$ etc. In the experiments described, the weights have been randomly initialised within $[-0.2, 0.2]$ or $[-0.1, 0.1]$, so those values can be considered huge, rendering the gates to be closed in the beginning, issuing not constant non-zero output to be used as fake-bias. Because the negative biases are decreasing over different blocks, they can reach a useful state one after another when assuming a rather small upper limit for the weight updates per iteration, which is valid for small learning rates.

5.2.1 Truncation

"Truncation" is described in the original LSTM paper as assuming certain derivatives to be zero when calculating the gradient. For a tree $t = (t_1, \ldots, t_\kappa, \ldots, t_k)$ the assumption is

$$\frac{\partial net_k(t)}{\partial o_i(t_\kappa)} = 0 \quad \forall k \in C_{in}, G_{in}, G_{out}.$$

For calculating the gradient, the terms

$$\frac{\partial o_k(t)}{\partial v_{ij\kappa}}, \frac{\partial o_k(t)}{\partial w_{ij}}$$

are being assigned variables (the error signals) only for $k \in C_s$ and only for weights going into an input gate or a cell input neuron of the according block, that is, $j \in C_{in}$ or $j \in G_{in}$. Their values are updated during the activation (more precisely during the recursive ascension of the neuron's activation) according to the formulas shown in the next chapter.

Roughly spoken the formulas are acquired by *assuming* only the neurons from C_s to be $\mathcal{R}$-Neurons as defined in 3.2.4 and thus ignoring error signals that would be $\neq 0$ otherwise.

5.2.2 Updating error signals ("forward pass")

Let m be any neuron, $s = c^i_{s,j}$ be a CEC, $g = g_{in,j}$ a gate neuron and $c = c^i_{in,j}$ a cell input neuron and let $\vec{1} = (1, \ldots, 1)^\top$ be the recursive weight of the CEC. The update of the error signals is then given below: actual indices for ϑ must be

derived from the weight. The expression $\vartheta = \vec{1} \bullet q^{-1}\vartheta$ neatly indicates the concept of constant error flow. For a neuron j let $f_j'[t] := f_j'(net_j(t))$:

$$\begin{aligned}
\vartheta_{m,j,i}(t) :&= \frac{\partial o_s(t)}{\partial w_{m,c}} \qquad \text{(feedforward to cell input: } m \to c^i_{in,j}) \\
&= \vec{1} \bullet q^{-1}\vartheta_{m,j,i}(t) + o_g(t) f_c'[t] o_m(t), \\
\vartheta^{(\kappa)}_{m,j,i}(t) :&= \frac{\partial o_s(t)}{\partial v_{m,c,\kappa}} \qquad \text{(recursive to cell input: } m \succ c^i_{in,j}) \\
&= \vec{1} \bullet q^{-1}\vartheta^{(\kappa)}_{m,j,i}(t) + o_g(t) f_c'[t] o_m(\chi_\kappa(t)), \\
\vartheta_{m,j}(t) :&= \frac{\partial o_s(t)}{\partial w_{m,g}} \qquad \text{(feedforward to gate: } m \to g_{in,j}) \\
&= \vec{1} \bullet q^{-1}\vartheta_{m,j}(t) + f_g'[t] o_c(t) o_m(t), \\
\vartheta^{(\kappa)}_{m,j}(t) :&= \frac{\partial o_s(t)}{\partial v_{m,g,\kappa}} \qquad \text{(recursive to gate: } m \succ g_{in,j}) \\
&= \vec{1} \bullet q^{-1}\vartheta^{(\kappa)}_{m,j}(t) + f_g'[t] o_c(t) o_m(\chi_\kappa(t)).
\end{aligned}$$

It is possible to avoid truncation and use full BPTS/BPTT or RTRL, but as mentioned in [HS97] for BPTT it did not qualitatively enhance the results.

5.2.3 Weight updates

All non-multiplicative weights can be trained as described below. The formulas for recurrent LSTM can be seen in [HS97, appendix A.1, "backward pass"], for recursive LSTM they are described in [Arn08, ch. 6.1]. Error signals are defined using backpropagation within the feedforward layer. A training example (t, y) with

$$t \in (\mathbb{R}^m)^+_k \,,\ y = (y_i)_{i \in O}$$

is assumed. It is $e_j(t) = \frac{\partial E(t)}{\partial net_j(t)}$ after truncation.

$$\begin{aligned}
e_j(t) &= f_j'[t]\,(o_j(t) - y_j(t)) && \forall j \in O, \\
e_j(t) &= f_j'[t] \sum_{k \in O} w_{jk} e_k(t) && \forall j \in H, \\
e_j(t) &= c_{i,v}(t) = \sum_{k \in O} w_{j,k} e_k(t) && \forall j = c^v_{out,i} \in C_{out}, \\
e_j(t) &= s_{i,v}(t) = o_{c_{out,i}}(t) f'_{c^v_{h,i}}[t] c_{i,v}(t) && \forall j = c_{s^v_i} \in C_s, \\
e_j(t) &= f_j'[t] \sum_{v=1}^{B_i} o_{c^v_{h,i}}(t) c_{i,v}(t) && \forall j = g_{out,i} \in G_{out},
\end{aligned}$$

having $1 \leq i \leq B$ (number of blocks), $1 \leq v \leq B_i$ (size of each block). Based on these error signals, the following effective weight updates $\Delta w(t)$ are computed:

$$\Delta w_{mj}(t) = -\alpha \begin{cases} e_j(t) o_m(t) & \forall j \in O \cup H \cup G_{out}, \\ \sum_{v=1}^{B_i} s_{i,v}(t) \vartheta_{m,j,v}(t) & \forall j = g_{in,i}, \\ s_{i,v}(t) \vartheta_{m,j}(t) & \forall j = c_{in,i}^v, \end{cases}$$

$$\Delta v_{mj}^{(\kappa)}(t) = -\alpha \begin{cases} e_j(t) o_m(\chi_\kappa(t)) & \forall j \in O \cup H \cup G_{out}, \\ \sum_{v=1}^{B_i} s_{i,v}(t) \vartheta_{m,j,v}^{(\kappa)}(t) & \forall j = g_{in,i}, \\ s_{i,v}(t) \vartheta_{m,j}^{(\kappa)}(t) & \forall j = c_{in,i}^v. \end{cases}$$

Therein the factor $-\alpha$ is already specified for usage in gradient descent. When the error function is to be maximised, the minus sign could of course be hold off.

5.3 Other training methods

"Simple weight guessing" was proposed in the original LSTM article [HS97] to distinguish between hard and easy problems and not as a proper training method itself. Instead of adapting parameters (weights) according to some algorithms, for example gradient descent, the freshly initialised net is evaluated on the training data. If the error is too high, the net is initialised again and the process is being repeated. Using this method, some learning tasks could be solved within several thousand repetitions which is a rather small number in comparison to the number of iterations usually needed for training. Other tasks could not be solved and the authors concluded that this tasks where sufficiently difficult.

In [BSF94, ch. 5] other training methods basing on re-randomising are mentioned (not related to LSTM).

A genetic training algorithm "EVOLINO" has been used for LSTM in [May+06] to produce trajectories respectively movement instructions for robot arms to form knots for surgical purposes (a process that is repetitive and time consuming for surgeons).

6 Restructuring of input data

In this chapter the concept of restructuring is introduced. After describing the motivation of this research, examples of previous work are given and one strategy is generalised onto trees. Several strategies to map sequences to structures are depicted. After giving an abstract overview of notions and pros and cons of the developed strategies, algorithms are presented that directly process sequential input data with a recursive net by parsing the input data as required for the respective restructuring method. Finally, the possibility of inversion for one certain method is examined.

6.1 Motivation

Recurrent neural networks process a given sequence from start to end, mapping each input vector together with the previous state vector into a new state vector. The ability to process input of dynamic length by the recurrent definition of the net's output can be seen as a trade-off between better generality and worse learnability due to upcoming long-term dependencies and fading gradient.

Considering a recurrent net as a compromise leads to the question, whether applying a recursive net in place of a recurrent net (by interpreting a sequence as a tree in a manner to be specified) could yield a gain due to a shortened maximum distance between single elements of the sequence and a local history that does not span the whole previous part of the sequence.

Of course, the need to find an appropriate structuring mechanism might constitute the first compromise and the potentially harder task to find a recursive property (to be simulated by the recursive net) where none can be expected might constitute the second.

On the other hand, when found, the recursive net could be used for rule-extraction. This means that a recursive rule for matching a given sequence is found that could help to understand the problem at hand more detailed because important activations within a subtree can only result from the data that is a part of this subtree.

Some of the structuring mechanisms used here are constructing regular trees where the elements of the sequence are the labels of leaf nodes while inner nodes have an implied label 0 (zero). This way the only time that the weight matrix connecting the input-layer with the context layer comes into effect is the activation of a leaf. This structuring methods can be understood as the instance of an *empty shell idiom* that can also be found on Hopfield nets (compare 6.2.1).

Multiple instances of the recursive net (unfolding) can also be interpreted as a swarm of (copied) agents that cooperate to form the mapping function.

Using this structuring mechanism also proposes a natural way of dealing with sparse data because no default node of some kind must be introduced to replace missing nodes but the initial context can be naturally used instead.

The application of artificial neural nets is a continuously developing research area. Network schemes like BRNN or contextual models using the q^{+1} operator are network structures that enlarge the context used for the neural net. The following mechanisms do not impose a special network architecture but offer several ways to enlarge the underlying context of sequential input for an arbitrary architecture by restructuring the input data into a tree.

Contrary to using the q^{+1} operator only the data itself is touched and the neural net structure remains untouched. Especially no further restrictions to the connection structure of a neural net are introduced because functional cycles via $q^{+1} \circ q^{-1}$ cannot occur. Each method can be combined with any neural net architecture as long as it is defined or definable as a recursive net.

Recursion in nature

The following example shall motivate the idea that recursive information processing is able to constitute a natural way of complex information processing where a setting is given that strongly limits the capacity for encoding the processing function in relation to the amount of data that is processed:

There are approximately 6 million cones and 120 million rods within the human retina. Assume a fly can perceive visual information at a rate of 200 Hz, that is, approximately the maximum firing rate of a biological neuron, without deep understanding of the surroundings but the ability to react fast on changes within it. Assume a human perceives visual input with deep understanding at a rate of only 20 Hz. Then one could assume that the $200/20 = 10$ time steps of primitive visual input are processed to a deeper understanding. If, however, this processing is done in 10 static layers then each layer could operate at a speed of 200 Hz and the final output would arrive with 200 Hz and just a *delay* of 1/20 s. This would

not explain the slow perception rate of 20 Hz. Assuming that some layers are not only receiving input from the cones and rods of the retina or neurons connected to them (feedforward input) but also from themselves (by recurrent connection) and that those layers need to collect several, for example 10, consecutive inputs for one output this drop rate would be explained. But as consecutive input of the *same* rods/cones or neurons would not offer the opportunity to recognise objects that span the whole visual field (the actual "deep understanding"), the interconnections must not only exist to themselves but also to neighbour parts, generating two or more recurrent inputs for one part, that is, making it an actual recursion ($k \geq 2$). Now the question arises how to encode the amount of at least 6 million $\times$ 6 million $= 36 \cdot 10^{12}$ connections into the human DNA which consists of only $3 \cdot 10^9$ base pairs. A possible answer is that for each layer only the structure of a single, small cluster is encoded and that during the growth of the actual layer this cluster is copied to a different location, creating "instances" of this cluster. Additionally, the same cluster could be used in consecutive layers, but with a bigger shift, that is, connections to clusters farther away. This process would end up in a structure comparable to the feedforward net that is acquired by unfolding a recursive net with the "recursive" restructuring mode as explained in section 6.4.1.

Though it is unlikely to achieve progress in research only with the naive and simplified argumentation above, dealing with recursive neural nets on sequential data might give insight to certain intrinsic properties that are reflected in real-world systems.

6.2 Existing work

In this section, certain novel forms of neural nets are interpreted as a restructuring method and existing work. Without being explicitly mentioned as such, a few mechanisms of restructuring have already been introduced in the literature. Bidirectional recurrent nets are probably the most recent example and Hopfield nets can also be seen as such. They will be described in the following.

6.2.1 Hopfield net

A Hopfield net is a recurrent net $h = (N, I, O, F, \mathcal{W}, \rightarrow, \succ, \emptyset, \vec{0})$ having $N = \{1, \ldots, 2n\}$, $I = \{1, \ldots, n\}$, $O = N \setminus I$, $\rightarrow = \{(i, i+n) | 1 \leq i \leq n\}$ with $v_{i,i+n} = 1$, $\succ = O \times O$ with $w_{ij} = w_{ji}$ and $w_{ii} \geq 0$. The architectural graph is found in figure 6.1. The input neuron's values are copied once into the hidden neurons who afterwards

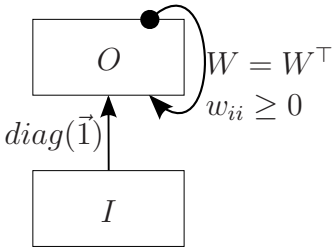

Figure 6.1. *Hopfield net*

are activated until the net function reaches a stable state. So the net is intended to map $x \in \{0,1\}^n$ to a fixed point by iterating the nets activation without inserting further labels into I:

$$f(x) = \lim_{j \to \infty} \mathcal{F}_h((x, \overbrace{\vec{0}, \ldots, \vec{0}}^{j \text{ times}})).$$

The activation functions $F = \{f_i = H | i \in N \setminus I\}$ are used where H is the Heavyside function

$$H(x) = \begin{cases} 1 & \text{if } x \geq 0, \\ 0, & \text{if } x < 0. \end{cases}$$

The limit must not always exist because cycles of length at most 2 can occur, but if existent it will be achieved within finite time due to the binary states of the net as shown in [Ham07]. When updating only one neuron's activation at a time, the limes can be shown to exist. The input neurons $1, \ldots, n$ are not defined in [Ham07] but are introduced in the above definition to make explicit that one vector of fixed length can be seen to be processed by a recurrent net that receives a sequence of vectors.

In an abstract view, this procedure maps the vector $x \in \{0,1\}^n$ to a linear chain $(x, \vec{0}, \ldots, \vec{0}) \in (\{0,1\}^n)_1^+$. The structure "vector" is formed into a "linear chain of vectors" where additionally created nodes do have an implied label 0.

6.2.2 BRNNs through Bidirectional Restructuring

The mapping function $f_{BRNN} : (\mathbb{R}^l)^+ \to (\mathbb{R}^m)^+$ of a BRNN (compare chapter 4.5) can also be acquired by applying a recursive Elman net (REN) with restrictions to the connection structure according to figure 6.2 on trees $T_k(x) = x_k(x_{k-1}(x_{k-2}(\ldots, \perp), \perp), x_{k+1}(\perp, x_{k+2}(\perp, \ldots))$ that are generated from the original sequence according to figure 6.3.

Let the REN be N and $\mathcal{F}_N$ its net function. It is then

$$(f_{BRNN}(x))_k = \mathcal{F}_N(T_k(x)).$$

The structure "linear chain" of x is formed into a "set of binary trees" T_k and the output of the BRNN at position k is now the same as the output of the REN on T_k.

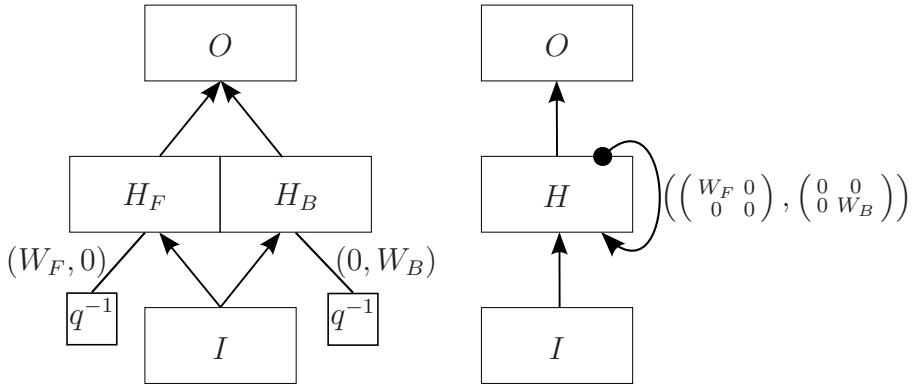

Figure 6.2. *Architectural graph of an BRNN that is transformed into a recursive net. The labels on the edges between the q-boxes restrict the architecture to have fanout* 2 *and to have weights* $\neq 0$ *only from the left child to the "forward" part of the hidden layer, that is,* H_F*, and from the right child to the "backward" part, that is,* H_B*.*

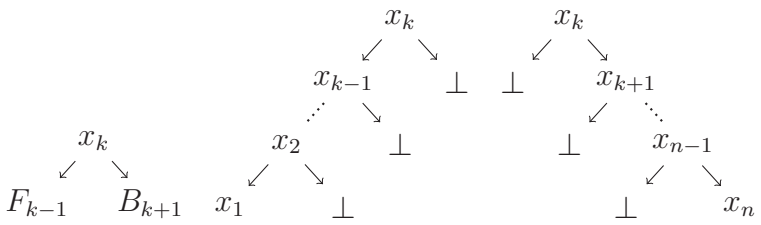

Figure 6.3. *BRNN-Tree* T_k *(left) consisting of the forward tree* F_{k-1} *(*F_k *displayed in the middle) and backward tree* B_{k+1} *(*B_k *displayed at the right).* $\perp$ *is used to clarify the positions.*

It is easy to see that a BRNN tree has more of the character of a sequence then a of a tree. In fact, the forward and backward trees are just sequences for which holds that they are modulated by their property of being "forward part" (left child) or "backward part" (right child).

The only difference between the transformed BRNN and a general REN is the restricted self connection of the hidden layer: in the unmodified definition of a REN those weights can be different from 0. It might be possible to reduce the restriction to a certain bias and initial context, but on the one hand, a BRNN

is only a motivation, and on the other hand, the ability to use a shared hidden layer is more practical, so no further research is done about this question. Using just a net without restrictions to the weights supersedes the need to find two different parameters for the architecture (the size of the forward and backward net, respectively). Furthermore it gives the opportunity to share neurons that might constitute computations that are symmetrical respectively invariant to the reversion of the order of the sequence elements.

Bidirectional LSTM

"Bidirectional LSTM" as introduced in [GS05] has been used to perform speech recognition. It is another form of bidirectional net where a larger "context" for a classification task is build for each output by taking future sequence elements into account within a separate hidden layer. The basic usage is equal to the definitions above but just using an LSTM network instead of an Elman net or a general MLP.

6.3 Convolution and Leaf Level Mirroring

In this section, bidirectional restructuring is generalised onto trees and accommodated for usage in classification tasks.

For such a classification task, only the output at the end of the input structure is compared against the target value. However, a BRNN would produce only output basing on the plain sequence, not even using the backward part of the net, even though within the sequence certain activations gathered from the backward part could correlate to the target. This drawback can be overcome by defining only one single tree for the recursive net to be processed. For this tree, each node in the sequence is enriched with additional data from its parents at a newly created position. This process can directly be generalised onto trees.

The general restructuring mechanism can be summarised as follows: for a given tree T with fan-out $k \in \mathbb{N}$ enlarge the fan-out to $k+1$ leaving every new position $k+1$ empty at first. Then for every subtree $U = u(v_1, \ldots, v_k, \perp)$ of T with empty position $k+1$ determine the labels of all its parents ascending to the root-label, build a sequence-like tree by placing the label of the parent as root-label, the label of the parent's parent at $k+1$-th position (leaving all other positions empty) and repeat until the root-label of T is used. Then place this generated tree at position $k+1$ of the subtree U. Formally, this procedure can be described by introducing a non-symmetrical convolution product between trees as follows:

Definition 12 (convolution product)**.** For a tree $T \in \Sigma_k^+$ and a subtree U with path $p(U) = (i_1, \dots i_D)$ within T at depth D let be $p_d(U) := (i_1, \dots, i_{D-d})$ the path to the d-th parent of U. Then define the labels reached on this path within another tree S by $\pi_d(U,S) := \lambda(\chi_{p_d(U)}(S)), 1 \leq d < D$ and $\pi_D(U,S) := \lambda(S)$. Then a tree consisting of these labels is defined by

$$\pi(U,S) = \pi_1(U,S)(\perp^k, \pi_2(U,S)(\perp^k, \pi_3(U,S)(\perp^k, \dots \pi_D(U,S)))) \in \Sigma_{k+1}^+.$$

For the tree T itself let $\pi(T,S) := \perp$. Using this definition a convolution product between trees $T, U \in \Sigma_k^+$ with $\mathrm{skel}(T) \subseteq \mathrm{skel}(U)$ is defined by

$$\begin{aligned} * : \Sigma_k^+ \times \Sigma_k^+ &\to \Sigma_{k+1}^+, \\ (T,S) &\mapsto T * S = T' \text{ having} \\ U' &= u(v_1, \dots, v_k, \pi(U,S)) \leq T' \\ &\Leftrightarrow U = u(v_1, \dots, v_k) \leq T. \end{aligned}$$

Using this convolution product, a tree T is recursively enhanced. For every subtree U all the labels from the corresponding subtree U' in S up to the root label of S are attached to the new introduced position.

Definition 13 (leaf level mirrored tree)**.** For $T \in \Sigma_k^+$ the tree $T * T \in \Sigma_{k+1}^+$ is called *leaf level mirrored tree of* T (LLM tree).

The LLM tree $X * X$ of a sequence X is depicted in figure 6.4. Obviously any such

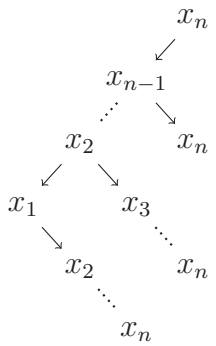

Figure 6.4. *This figure shows what $X * X \in \Sigma_2^+$ looks like if a sequence $X \in \Sigma^+$ is interpreted as an element of Σ_1^+. Compare figure 6.3.*

tree $T * T$ contains redundant information so an algorithmic implementation for training a recursive net on $T * T$ should avoid temporarily generating this structure prior to activating it.

As the attached trees within a LLM tree are sequence-like and have common subsequences, they can be merged. This creates a directed ordered acyclic graph where common subsequences are merged into the same node and the lower half of this graph is roughly acquired by positioning all leaves at the same (height) level and mirroring the upper half at this "leaf level".

This method of restructuring would also properly be named by "root path attachment", however this suggests the existence of one unique root. Without formalisation, the notion of mirroring allows to generalise this method to directed ordered acyclic graphs. A structure with maximum fan-out k_1 and fan-in k_2 could thus be mapped to a mirrored structure with maximum fan-out $k_1 + k_2$ and fan-in k_2.

6.4 New strategies

In this section, several strategies for mapping sequences to structures are proposed.

There are many ways to map a given sequence $x = (x_i)_1^n$ to a tree. For the experiments conducted in this thesis, the following methods have been defined and compared:

1. The given sequence is supposed to be the set of labels within the leaves of a complete k-ary tree where non-leaves have the implied label 0. For sequences of length $\neq k^n,\ n \in \mathbb{N}$, the tree is reduced at the right side such that subtrees which do not contain leaves of the deepest level (i.e. none of its leaves contains a vector from the sequence) are removed.

   This mode keeps the topological order of the sequence intact in the sense that "x_i is a left neighbour of x_j" holds if $i < j$.

   It is referenced as "recursive" for short.

2. The sequence is formed into a set $\{T_i(x)|0 \leq i \leq n-1\}$ of n binary trees of depth $D := \lceil log_2 n \rceil$ with $n' := 2^D$ leaves into each of which the sequence elements are inserted, starting from position i, while inner labels are $\vec{0}$:

   $$T_i(x) := \vec{0}\Big(\dots \vec{0}\Big(x_{1+i}(), x_{2+i}()\Big) \dots \ , \ \dots \ \vec{0}\Big(x_{n'-1+i}(), x_{n'+i}()\Big) \dots \Big).$$

   The indices are assumed to be modulo n, which is plausible for a periodic pattern of infinite length. Every (finite) sequence can be interpreted as such by repeating it infinitely.

   This mode is referenced as "periodic" for short.

3. To create a binary tree, the middle of the given sequence is taken as the root-label and the left and right part of the sequence are processed recursively to form the two subtrees that are the left and right child of the root. The structure is unique for sequences of length $2^n - 1, n \in \mathbb{N}$. Otherwise the rounding mode determines the concrete structure. This can be understood as a Divide and Conquer strategy where the processing of two previously divided parts is done by a recursive net. Training the net actually means to find an appropriate joining mechanism.

   As this mode turns out to be some kind of Divide and Conquer strategy, it is referenced as "Divide and Conquer" (D&C) for short.

4. As the D&C mode generates a tree from a sequence and bidirectional nets can be generalised onto trees, both strategies can be combined into Bidirectional Divide and Conquer.

Note that for "periodic" and "recursive", missing parts of the given sequence can be modelled as $\perp$ instead of a child $x_0()$ containing a default vector x_0. As a consequence, if the initial context is defined as a fixed point of the transition function by means of $x = f(\vec{0}, x, x)$, contiguous gaps result in the ability to delete all subtrees spanning over elements within this gap. Even without a fixed point, activations of the same depth within these subtrees can be reused so that only one activation for each depth must be computed.

The proposed modes will be described mathematically in the next sections and some pros and cons of each mode will be addressed afterwards.

6.4.1 Recursive

The recursive restructuring mode is defined by making use of a recursive helper mapping and has the parameter $2 \leq k \in \mathbb{N}$. Let $x \in \Sigma^n$ be a sequence and $\vec{0} \in \Sigma$. Define

$$y_i^0 := x_i(\perp^k) \in \Sigma_k^+ \ \forall 1 \leq i \leq n.$$

Because this method constructs a tree with leaves all at the same depth, the term "height" shall be introduced where $h = D - d$ is the height of a certain subtree of depth d and D the maximum depth. Let $D := \lceil \log_k n \rceil$ so that k^D is the length of x rounded up to the next power of k and recursively define a mapping that constructs a tree that has y_i^0 as labels at height 0 and $\vec{0}$ as labels at height > 0. For doing so, a new subtree y_i^h at height h is build if and only if it can contain a part of the

sequence, that is, if the number of leaves of all subtrees with same height to its left does *not* suffice to cover the whole sequence, that is, when $(i-1)k^h < n$:

$$y_i^{h+1} := \vec{0}\left(\tilde{y}^h_{(i-1)k+1}, \ldots, \tilde{y}^h_{(i-1)k+k}\right) \in \Sigma_k^+$$
$$\forall h \in \mathbb{N}_0, i \in \mathbb{N} : h < D, i < n/k^h + 1$$

with

$$\tilde{y}_i^h = \begin{cases} y_i^h & \text{if } i < n/k^h + 1, \\ \perp & \text{if } i \geq n/k^h + 1. \end{cases}$$

Note that by definition of D the root of y_1^D always has at least two children $y_{1,2}^{D-1}$ because otherwise $n \leq k^{D-1}$ would hold. The recursive mapping mode is then defined by

$$(6.1) \quad {}^{(r)}\frown : \Sigma^+ \to \Sigma_k^+,$$
$$x = (x_1, \ldots, x_n) \mapsto {}^{(r)}\widehat{x} := y_1^D.$$

Examples are given in figure 6.5 for $\Sigma = \mathbb{R}^l$.

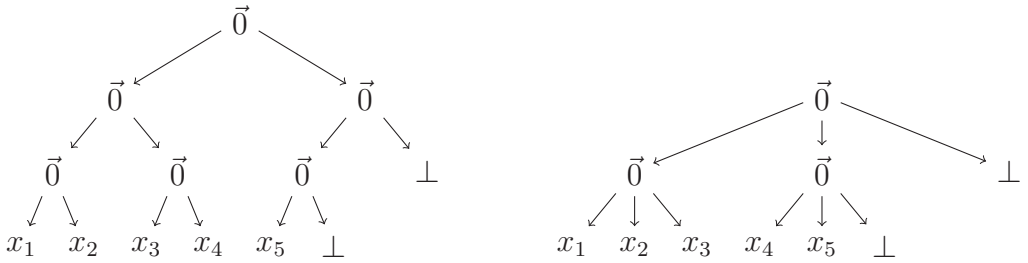

Figure 6.5. *Graphical example for the recursive restructuring of* $x = (x_1, \ldots, x_5) \in \left(\mathbb{R}^l\right)^+$ *to* ${}^{(r)}\widehat{x} \in \left(\mathbb{R}^l\right)_2^+$ *with* $k = 2$ *and and* ${}^{(r)}\widehat{x} \in \left(\mathbb{R}^l\right)_3^+$ *with* $k = 3$.

There is an evident correspondence between the last non-empty position per height and the k-adic representation of numbers. This is due to the fact that knowledge of the last non-empty position allows to nest the estimated amount of missing leaves into decreasing powers of k. More precisely: let l be the leaf containing the last element of the input sequence x and $p = (1 + p_1, \ldots, 1 + p_D)$ the path of l within ${}^{(r)}\widehat{x}$. It is $l = y_n^0$ with an unknown $n \in \mathbb{N}$. Then the k-adic representation of $n - 1$ is $p_1 \ldots p_D$.

Leaf-level mirroring after recursive restructuring

Remark that leaf-level-mirroring could be applied to the result of the recursive restructuring so that a pattern $m'' := ({}^{(r)}\widehat{x} * {}^{(r)}\widehat{x}, y)$ is created. However, this introduces a third position that does not contain any data related to the original sequence x. Each new position at depth d simply contains the d-th iteration $\vec{0}(\perp, \perp, \vec{0}(\ldots))$ of the initial context. When using a recursive net to learn m'' one could of course aim for a special initial context and weights W_3 for the third position so that the net computes predefined values $v(d) \in \mathbb{R}^l$ for each specific depth that might improve the learning task. However these values could also be directly inserted by modifying the recursive mode to use $\vec{0}; v(d)$ for inner labels and $x_i; v(D)$ for the leaves.

Presumably the only reason to apply leaf-level mirroring would be to have the net automatically learn a height-dependent virtual initial context for the new third position.

Alternative method

As an alternative to this method, a sequence of length $\neq k^D$ could be restructured by trying to create k-regular subtrees at different heights. For $k = 2$ this would result in the fact that empty positions would only occur at leaves $x(\perp, \perp)$ and not as single empty positions like in $x(y(\ldots), \perp)$. This could be helpful for tasks where the initial context is of special interest because for a recursive net with weight tensor (W_1, W_2) it influences the activation via $\xi(W_1 + W_2)$. But as a side effect the height or depth of all leaves is not the same anymore by this method and therefore it has not been examined any further.

6.4.2 Periodic

The periodic restructuring mode is comparable to the recursive mode but it creates as much trees as the input sequence is long and assumes the actual data to be a periodic series as $x; x; \ldots$ of which the sequence x is only one repetition. It also uses a recursive helper mapping and has the parameter $2 \leq k \in \mathbb{N}$. Let $x \in \Sigma^n$ be a sequence and $\vec{0} \in \Sigma$. Define

$$y_i^0 := x_i(\perp^k) \in \Sigma_k^+ \ \forall 1 \leq i \leq n.$$

Again let $D := \lceil \log_k n \rceil$ so that k^D is the length of x rounded up to the next power of k. Now k-regular trees of depth D are constructed over the elements of x, starting

at the i-th component for $1 \leq i \leq n$, generating a total of n trees. The following recursive mapping is defined:

$$(*) \qquad y_i^{h+1} := \vec{0}\left(\tilde{y}^h_{i+0\cdot k^h}, \ldots, \tilde{y}^h_{i+(k-1)k^h}\right) \in \Sigma_k^+ \qquad \forall h \in \mathbb{N}_0, i \in \mathbb{N} : h < D, i \leq n.$$

This time accessing elements behind the n-th component are mapped back into the respective level:

$$\tilde{y}_i^h := y^h_{i \bmod n}.$$

The representative of the residual class [0] modulo n is chosen to be n so that indices range between 1 and n. Equation $(*)$ bases on the idea of defining a k-regular tree T with k^h leaves x_{i+0}, …, x_{i+k^h-1} by accessing k k-regular trees T_κ with leaves $x_{i+(\kappa-1)k^{h-1}}$, …, $x_{i+\kappa k^{h-1}-1}$. Making each level periodic by accessing modulo n is shown to be correct by induction based on y_i^0 to be periodic modulo n. The recursive mapping mode is now defined by

$$(6.2) \quad {}^{(p)}\frown : \Sigma^+ \to \left(\Sigma_k^+\right)^+,$$
$$x = (x_1, \ldots, x_n) \mapsto {}^{(p)}\widehat{x} := (y_j^D)_{j=1}^n.$$

For a given pattern $m = (x, y)$, a recursive net N can then be trained on a set of patterns $\{m' = (x', y) | x' = ({}^{(p)}\widehat{x})_j, 1 \leq j \leq n\}$. Though the number of patterns increases by the factor n, the computational complexity increases only by $\log_k n$ if the plenty common partial results are shared (see 6.6.2). If x is of length exactly k^D, the following relation between recursive and periodic restructuring holds:

$$z_j := (x_j, \ldots, x_n, x_1, \ldots, x_{j-1}) \Rightarrow {}^{(r)}\widehat{z_j} = ({}^{(p)}\widehat{x})_j.$$

The sequences z_j are the result of cyclic movement of the elements of x. The trees ${}^{(r)}\widehat{z_j}$ are their recursive restructured counterparts. Obviously they can be merged into one directed, ordered, acyclic graph where the nodes y_i^h are the same height h and each node except for the lowest level has k outgoing connections by definition. Compare figure 6.6.

Pairwise different childtrees?

The childtrees have incoming connections from $y_{j_\kappa}^{h+1}$ with $j_\kappa = i - \kappa k^h$. They are not necessarily pairwise different so that some y_i^h serves multiple times as a childtree (at different positions) for the same y_j^{h+1}. This can be seen by assuming $j_\kappa = i - 0$:

$$i + \kappa k^h \equiv i \bmod n \Leftrightarrow \kappa k^h \equiv 0 \bmod n \Leftrightarrow n | \kappa k^h.$$

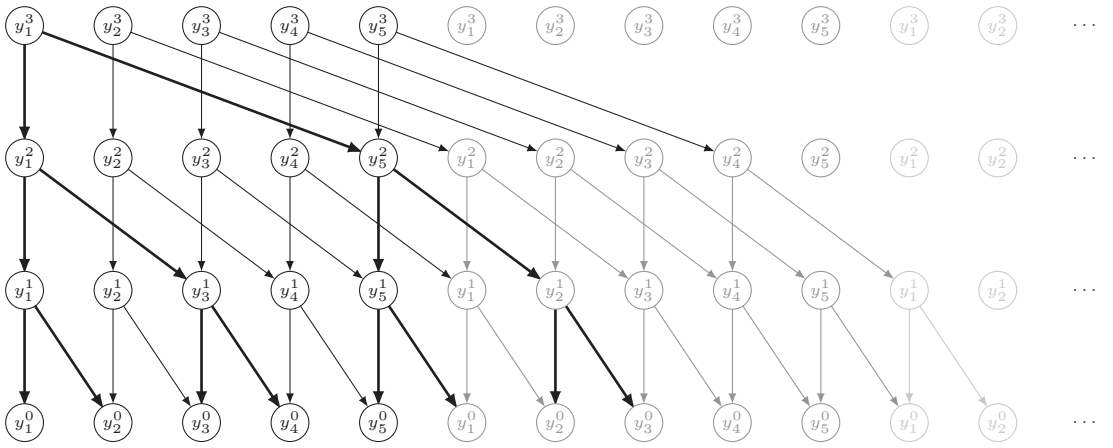

Figure 6.6. *The trees in* ${}^{(p)}\widehat{x}$ *merged into one graph for an input sequence of length* $n = 5 \leq 2^3$ *that results in* $D = 3$*. Contrary to other graphical displays of trees within each circle the name of the corresponding (sub)tree is shown. Each level is periodically expanded and each repetition is depicted as fading to make the repetition obvious. An algorithmic implementation of this mode only requires to store the first five nodes of each layer. The tree* y_1^D *is depicted in thick lines. It does not contain gray lines though spanning over gray nodes: all subtrees of* y_1^D *that have a gray root would need to be created separately if not the whole structure had been merged together with* $y_2^D, \ldots, y_5^D$*; for example the subtree* y_2^1 *in black nodes is not part of* y_1^D *but of* y_2^D *so that within* y_1^D *it can be replaced by a pointer into* y_2^D*.*

It is $k^{D-1} < n \leq k^D$. At first assume $n = k^D$: then the biggest stride $(k-1)k^{D-1}$ is too small to be a multiple of n so the equation holds only for $\kappa = 0$. For a fixed parameter $k = 2$ this equation also holds only trivially because the only other possible value $\kappa = 1$ still keeps the stride to small: $1 \cdot k^{D-1} = k^{D-1} < n$.

But in general, for example for $k = 3$ and $n = 6 = 2 \cdot 3^1$ the equation can hold (here using $\kappa = 2$) and on the top-most level each y_i^D makes use of y_i^{D-1} at more than one position (here: position 1 via $\kappa = 0$ and position 3 via $\kappa = 2$). For short sequences with $n < k$ this obviously holds, too.

This fact must not present a problem to the restructuring method itself but may be unwanted for applications. For example, when applying a recursive Elman net with weight tensor (W_1, W_2) to the input structures ${}^{(p)}\widehat{x}$, the layer-transition function can be described by $y_*^h \mapsto y_*^{h+1} = f(W y_*^h)$ with a block matrix $W \in \mathbb{R}^{(hn)^2}$ (compare section 6.6.2.1, page 79). The blocks are of the size of $h \times h$. If periodic restructuring is used with $k = 2$, only the blocks 0, W_1 and W_2 can occur. But in general, additional blocks $W_1 + W_\kappa$ can occur in W which could make questions regarding mathematical properties more inconvenient to handle.

6.4.3 Divide and Conquer

The D&C restructuring mode creates a binary tree by selecting the middle element x_M of a given sequence $x = (x_i)_{i=a}^{b} \in \Sigma^n$ as the root label using $M = (a+b)/2$ rounded to an integer value; the child trees are recursively created out of $(x_a, \ldots, x_{M-1})$ respectively $(x_{M+1}, \ldots, x_b)$. This restructuring mode is parameterised by the underlying rounding function $r : \mathbb{R} \to \mathbb{Z}$ and formally defined as follows:

$$\begin{aligned} {}^{(m)}\frown &: \Sigma^* \to \Sigma_2^*, \\ \bot &\mapsto \bot, \\ x = (x_1, \ldots, x_n) \mapsto {}^{(m)}\widehat{x} &:= x_M \left({}^{(m)}\widehat{(x)_1^{M-1}}, {}^{(m)}\widehat{(x)_{M+1}^{n}} \right), \\ M &:= r((n+1)/2) \end{aligned}$$

with the convention $(x)_i^j = \bot$ if $i > j$ and $\Sigma^* := \Sigma^+ \cup \{\bot\}$. The rounding mode $r(x) = \lfloor x \rfloor$ has been used for the examples in figure 6.7. It matches the used

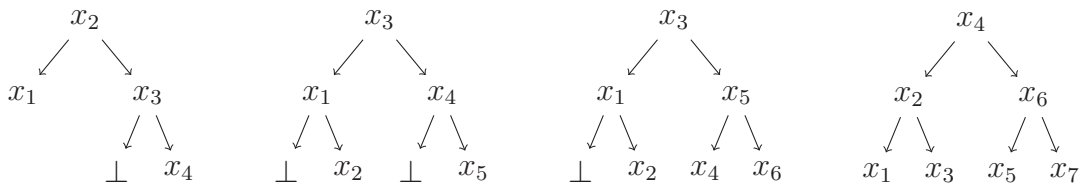

Figure 6.7. *Graphical example for the D&C restructuring of sequences of length* 4 *to* 7 *from* $\left(\mathbb{R}^l\right)^+$ *to* ${}^{(m)}\widehat{x} \in \left(\mathbb{R}^l\right)_2^+$ *with* $r(x) = \lfloor x \rfloor$*. A regular tree is created if the sequence has the length* $1\ldots1_2 = 2^n - 1$*.*

rounding mode for the conducted experiments. The following properties can be seen in this example:

- Starting at a subtree and following a path only over the 1st (left) position leads to monotonic decreasing indices.
- Following the 2nd (right) position leads to increasing indices.
- Child and parent node indices differ the more the closer they are to the root.
- Considering a sequence of sequences $y_1 = (x_1), y_2 = (x_1, x_2), \ldots$ which are respectively the prefix of each other, the relative position between x_l and x_k within the trees ${}^{(m)}\widehat{y_j}$ does not stay the same for fixed l, k and varying j.

Especially the last property makes this mode seem to act somewhat arbitrary on the input data. Nevertheless the conducted experiments will show that this mode is the most robust one.

Bidirectional Divide and Conquer

The concept of bidirectional restructuring, leaf-level mirroring, can be applied to any tree, including ${}^{(m)}\widehat{x}$. Therefore by creating ${}^{(m)}\widehat{x} * {}^{(m)}\widehat{x} \in \Sigma_3^+$ the restructuring mode "bidirectional Divide and Conquer" is defined.

6.5 Applications, advantages and drawbacks

The mentioned restructuring methods present novel ways for classifying data using neural nets. They also yield the opportunity for other processing tasks, for example:

- Error correction for sequences of varying length by training $\tilde{x}_1^n \mapsto x_1^n$ with $\tilde{x}$ being the sequence with disturbed or missing elements.
- Aligning of non translation invariant, periodic pattern by training $x_1^n \mapsto (i/(n-1))_{i=0}^{n-1}$ as a sequence-to-sequence mapping.
- A non-causal sequence-to-sequence prediction, for example for training a sensor array of n communicating sensors that use only $\log n$ computing time to predict their next state depending on *all* other sensors.
- The "non-causal sequence-to-sequence prediction" makes it possible (however not necessary feasible) to learn context sensitive grammars by presenting a sequence $x = (x_i)_{0 \le i < 2^D}$ and specifying for every contiguous subsequence $x(i,d) := (x_{i+j \bmod 2^D})_{0 \le j < 2^d - 1}$ with $0 \le i < 2^D$ and $0 \le d \le D$ of length 2^d whether it is an accepted word or not. Of course the fact that acceptance can only be specified for words of length k^d (as with $k = 2$) is a drawback. But this problem could be approached by replacing the inner labels $\vec{0}$ with indicators defining whether the underlying sequence part is the end, the beginning or an inner part of an accepted word.

Technical advantages

The distance from occurrence of the initial context (and every occurring input label) is drastically shortened from n to $log_k n$ so that long term dependencies can be expected to be less of a problem.

Using a recursive net offers the chance of parallelization, since every node within a layer of same depth within a tree can be activated depending only on nodes of

the layer below. The computational task is of a lesser fine granularity than, for example, computing the net input for a single neuron. Thereby it is well suited for many-core systems, for example. Additionally, the maximum possible speedup relates to the size of the input structure (not taking into account the requirements for memory access).

Technical drawbacks

Using neural nets for controlling tasks is possible, but standard methods for analysing the behaviour of the resulting controller usually fail. When periodic restructuring has been used, this property is worsened: if the impulse response of a recursive net (using the periodic mode) is analysed by activating a sequence $x_n := (0^{(n-1)}, 1, 0^{(n)})$ of length $2n$, the whole output sequence of length $2n$ differs in general and it cannot be guaranteed that $\mathcal{F}_N({}^{(p)}\widehat{x_n})$ is a subsequence of $\mathcal{F}_N({}^{(p)}\widehat{x_m})$ for $m > n$. This however should probably hold for a controller, as increasing the resolution of the discrete time readings (i.e. increasing m) should not severely affect the controllers behaviour.

The non-causal sequence-to-sequence mapping

Using the periodic mode the opportunity for a computationally inexpensive method for classifying a periodic pattern is given (see below). When classifying the same sequence in another order, that is, after shifting the window over the period, many previously done activations can be re-used. While for a periodic pattern the aforementioned method requires specifying the exact same target regardless of the starting position, it also gives the opportunity to assign a vector explicitly depending on the shift. This means nothing else than mapping a sequence to a sequence which can also be understood as an offline time series prediction. This could be used for training error correction codes on a data stream $D(n)$ by using parity data $P(D(n))$ that is mixed into the data stream before the time point n. The result is a non-causal relationship between the data stream elements $D(n)$ because changing data at time n changes the parity data at a time $n' < n$. Such a task could not be learned with a recurrent net.

Understanding the sequence-to-sequence mapping as an offline time series prediction renders the user able to apply machine learning on systems that have no deterministic behaviour at a single specific point, but that could deterministically be extrapolated if future inputs were already known.

Translation invariance

Tasks may occur where a sequence must be classified regardless of where the processing starts.

i) On the one hand this can hold for example for cyclic genes that are represented by encoding the nucleotide or amino acid sequences starting at an *arbitrary* position and stopping when reaching this position again. The encoding sequence is a *representative* for all sequences that would have been created by starting at another position.

ii) On the other hand a sensor array could be arranged on a cyclic structure like a circle or torus that contains an entity that is not equally distributed within this structure. The evaluation or classification of the sensor readings however would be expected to be *implicitly aligned* so the result is the same regardless of how the entity was oriented.

Let

$$\begin{aligned} c :& M^+ \to M^+, \\ & (x_a, \ldots, x_b) \mapsto (x_{a+1}, \ldots, x_b, x_a) \end{aligned}$$

be an operator that moves every element within a sequence to the left and the first element to the last position. For a sequence of length n it is $c^n = c^0 \equiv \mathrm{id}$. Using this operator the previous examples can be translated into the following problem:

i) Let $M \subset P$ be a subset of all possible patterns P that is to be classified. Then with $c^*M := \bigcup_{n\in\mathbb{N}_0}\{c^i(m)|m \in M\}$ let $c^*M \subseteq P$ hold. This case represents a *translation invariant pattern set*. For a classifier $f : P \to \{0, 1\}$ must hold: $f(c^i(m)) = f(c^j(m))\forall i, j \in \mathbb{N}_0, m \in M$. This can be achieved for example by training on $M' := c^*M$.

ii) Let $f : P \to \mathbb{R}$ be a mapping. Then

$$f^c : (x_1, \ldots, x_n) \mapsto \frac{1}{n}\sum_{i=1}^{n} f(c^i(x))$$

defines a *translation invariant mapping*.

Both cases require the computation of $f(c^i(x))$ for each $0 \leq i < n$. If the computational costs for computing $f(x) = \mathcal{F}_N(x)$ are $\mathcal{O}(g(\eta))$ then the costs for all $c^i(x)$ are $\mathcal{O}(\eta g(\eta))$. The computational costs are multiplied by η. If however the periodic restructuring mode is applied, the values $f(c^i(x))$ generate computational costs that grow only by the factor $\log(\eta)$.

The difference in learning between the two mentioned cases is the impact they have on the error function (compare chapter 3.1). While for case i) just a bigger pattern set $c^*M \supset M$ is used with the same pattern-wise error function $E = \sum_{m \in c^*M} E(m)$, the case ii) requires to keep using $E = E(M) = \sum_{m \in M} E^C(m)$ with a new pattern-wise error function $E^C(m)$ that can be defined for example by replacing $f(x)$ with $f^c(x)$. The gradient computation for E^C can easily be done by applying the original error function (for example E_{CE} or E_{SSE}) on the average output f^c:

$$\frac{\partial E^C(f^c(x))}{\partial w} = \frac{\partial E(f^c(x))}{\partial f^c(x)} \left(\frac{1}{n} \sum \frac{f(c^i(x))}{\partial w} \right).$$

When using the cross entropy E_{CE} it might be more consistent to use the geometrical mean for f^c instead of the arithmetical mean.

6.6 Algorithms for implicit restructuring

In this section algorithms are described to implement the restructuring modes on a given sequence without the need to temporarily create the defined structures. For this purpose, the transition function f of the (biased) general recursive net N will be used and as a conclusion, the algorithms compute the *state* of N within the restructured input T and not the actual output $\mathcal{F}_N(T)$. As mentioned in chapter 4.1, the mapping from the state of N within T to $\mathcal{F}_N(T)$ must be adapted to the actual architecture that is used. Here this is explicated for periodic restructuring for Elman nets.

6.6.1 Recursive

Given a (biased) recursive net N, its transition function f and a sequence $x \in \left(\mathbb{R}^l\right)^+$. Then the state of N within ${}^{(r)}\widehat{x}$ can be computed without recursive function calls according to algorithm 1 where indices are assumed to start with 1. The allowed fan-out k for the net is determining the free parameter k of the restructuring method. The function push_with_carry is implemented according to algorithm 2. Remark that updating of error signals for RTRL can be done while activating the net by calling f. Therefore, these algorithms implement a traversing method of the tree ${}^{(r)}\widehat{x}$ like mentioned in 3.2.3 using $log_k(n)$ memory.

When a sequence of length k^D is processed, it is $P.size() = D$ with $P[1].size() = k$ and $P[i].size() = k - 1$ for $i > 1$ after the first for-loop. It *prepares* the states to be stored in P while the next for-loop does *follow-up* work. Even though no extra

Algorithm 1 Recursive Restructuring

Require: transition function f of N, initial context ξ, sequence $x \in \left(\mathbb{R}^l\right)^n$
Ensure: y state of N within ${}^{(r)}\widehat{x}$

initialise vector of vectors P ▷ vector of vectors = ${y^d}_i$ only at the right end
P.resize(1)
for $i = 1 \ldots n$ **do** ▷ create inner states iteratively
    $fx \leftarrow f(x[i]; \xi; \ldots; \xi)$ ▷ state within leaf with label $x[i]$
    $P \leftarrow$push_with_carry(fx,1,P)
end for
for $h = 1 \ldots P.size() - 1$ **do** ▷ finalise by attaching $\perp$ excluding top-most level
    $\kappa \leftarrow P[h]$.size()
    $fx \leftarrow f(\vec{0}; P[h][1]; \ldots; P[h][\kappa]; \xi; \ldots; \xi)$
    $P \leftarrow$push_with_carry(fx,$h+1$,P) ▷ can increase P.size()!
end for
$h \leftarrow P$.size(), $\kappa \leftarrow P[h]$.size()
$y \leftarrow f(\vec{0}; P[h][1]; \ldots; P[h][\kappa]; \xi; \ldots; \xi)$ ▷ error injection here for learning
return y

Algorithm 2 push_with_carry

Require: state vector fx, height L, states P, transition function f
Ensure: P modified states according to new leaf-state fx.

repeat
    have_carry←false ▷ initialise: at first, no carry
    if $P[L]$.size() $= k$ **then** ▷ level would overflow
        carry← $f(\vec{0}; P[L][1]; \ldots; P[L][k])$ ▷ so activate first
        have_carry←true ▷ remember to push carry later
        $P[L]$.clear() ▷ level clear for pushing fx
    end if
    $P[L]$.push_back(fx) ▷ the actual intention of this function
    if have_carry=true **then** ▷ prepare the next loop
        $fx \leftarrow$carry, $L \leftarrow L + 1$
        if $L \geq P$.size() $+ 1$ **then** ▷ dynamically increased up to D
            P.resize(L)
        end if
    end if
until have_carry=false ▷ stop when no overflow could have occurred
return P

$\perp$ will be attached in the second loop in the case $n = k^D$, each iteration of the loop does important computations: it successively performs an overflow procedure that correlates to computing $1 \dots 1_2 + 1_2 = 10 \dots 0_2$ in binary. Because of this it is worthwhile to consider the *amount* of leaves, equally starting indices with 1, instead of considering the position of a leaf with indices starting at 0 (compare k-adic representation in 6.4.1).

6.6.2 Periodic

Given a (biased) recursive net N, its transition function f and a sequence $x \in \left(\mathbb{R}^l\right)^+$. Then the state of N within $x' \in {}^{(r)}\widehat{x}$ can be computed without recursive function calls according to algorithm 3. Therein the indexing starts with 0 to stick to the default implementation of the modulo operator "%". The variable D should be computed with integer arithmetic, as using floating point arithmetic may result for example in $\lceil \log_5 125 \rceil = 4$ depending on the hardware. The allowed fan-out k for the net is

Algorithm 3 Periodic Restructuring

Require: transition function g of N, initial context ξ, sequence $x \in \left(\mathbb{R}^l\right)^n$
Ensure: $y := L[D]$ states of N within ${}^{(p)}\widehat{x}$

```
Initialise array of arrays L                          ▷ indices 0...D and 0...n−1
D ← ⌈log_k n⌉
for d = 0...D do                          ▷ D+1 layers, the first for encoding labels
    for i = 0...n−1 do
        if d=0 then
            L[d][i] ← g(x[i]; ξ; ...; ξ)                          ▷ state of N in a leaf
        else
            stride← k^(d−1)
            arg← 0⃗
            for κ = 0...k−1 do                    ▷ successively concatenate vectors
                arg←arg; L[d−1][(i + κ · stride)%n]          ▷ % is modulo operator
            end for
            L[d][i] ← g(arg)                          ▷ error injection here if d = D
        end if
    end for
end for
return L[D]
```

determining the free parameter k of the restructuring method. Again, updating of error signals for RTRL can be done while activating the net by calling f and this algorithm shows one unique advantage of RTRL over BPTT respectively BPTS:

n trees can be not only activated but also learned (by specifying the same target n times or n different targets) with memory complexity $\mathcal{O}(n)$ and computational complexity $\mathcal{O}(n \log n)$. Contrary, BPTS requires to store all activations, resulting in memory complexity $\mathcal{O}(n \log n)$ and to learn for every individual $({}^{(p)}\widehat{x})_j$ by tracing back the path from the root to every leaf because the activations within each root differ in general. This means that paths of error backtracking that meet at a joined node y_i^d will result in different updates for the respective error signals due to the different origins of the paths that backtracking has started with. This results in $\mathcal{O}(n^2 \log n)$ computational costs. Of course, in practise and for a fixed architecture size BPTS can still be faster up to a certain length n_0 because of RTRLs update costs growing quadratically with respect to the number of weights. This of course does not hold for LSTM when using its special learning algorithm.

For learning, of course the target y may differ for each position j, so for example a periodic time series could be learned.

The error function used for the periodic mode must be adapted in the case that translation invariant patterns are assumed. Having such patterns can however always be assumed if for example a dedicated symbol is attached to each input sequence. This is done for the conducted experiments (compare "terminator symbol", page 94). As a conclusion, the following error function is used for the periodic mode, basing on the operator c^* as defined at page 75:

$$E = \frac{1}{|M|} \sum_{m \in M} \frac{1}{|c^*\{m\}|} \sum_{m' \in c^*\{m\}} E(m').$$

It results from collecting the cycled sequences $c^*\{m\}$ into one group that contains as many patterns as the original pattern m is long, that is, $|c^*\{m\}| = 1 + |\operatorname{skel} m|$. Each group is normalised (scaled) and then collected into the main group that is scaled again, compare chapter 3.4.

6.6.2.1 Special case for Elman nets

Given a sequence $x \in (\mathbb{R}^l)^{2^D}$ of length $n = 2^D$ and the fixed parameter $k = 2$. Let N be an Elman net with hidden layer H, $|H| = h$ with activation function f, input-to-hidden weight matrix $I \in \mathbb{R}^{h \times l}$ and recursive weights $W_1, W_2 \in \mathbb{R}^{h \times h}$. Let O denote the weight matrix from the hidden layer to the linearly activated output layer. For brevity the bias and initial context are assumed to be 0.

Then the whole mapping procedure of algorithm 3 regarding the transition function g can be reduced to a sequence of matrix multiplications and pointwise application of f. The weight matrices W_1 and W_2 determine the state transition. Let $y_i^d = L[d][i] \in \mathbb{R}^h$ from algorithm 3 and $y_*^d := y_1^d; \ldots; y_n^d = L[d] \in \mathbb{R}^{hn}$. It is then recursively

$$y_*^d = f(W(d)y_*^{d-1}), \quad y_*^0 = f(Ix_1); \ldots; f(Ix_n). \tag{6.3}$$

Note that W is not transposed in this notation. The matrix $W = W(d)$ consists of blocks W_1 and W_2, the weights of N. Their positions vary for each depth d. Each row of W reflects one value of i of the according for-loop. According to code lines 10 to 12 the blocks can be merged into bigger blocks A and B consisting of 2^{d-1} sub-blocks due to the stride 2^{d-1}:

$$A := \begin{pmatrix} W_1 & & 0 \\ & \ddots & \\ 0 & & W_1 \end{pmatrix} = diag(\overbrace{W_1, \ldots, W_1}^{2^{d-1} \text{ times}}), 2^{d-1} \text{ blocks of size } h,$$

$$B := \begin{pmatrix} W_2 & & 0 \\ & \ddots & \\ 0 & & W_2 \end{pmatrix} = diag(\overbrace{W_2, \ldots, W_2}^{2^{d-1} \text{ times}}), 2^{d-1} \text{ blocks of size } h,$$

$$W = \begin{pmatrix} A & B & 0 & \cdots & 0 \\ & \ddots & \ddots & & \\ 0 & \cdots & 0 & A & B \\ B & 0 & \cdots & 0 & A \end{pmatrix}, 2^{D-(d-1)} \text{ blocks of size } h2^{d-1}.$$

The parameter d (i.e. the current height within the structure), is a parameter to the weight matrix $W = W(d)$. As an example, for $n = 4$ the following block matrices are formed:

$$W(1) = \begin{pmatrix} W_1 & W_2 & 0 & 0 \\ 0 & W_1 & W_2 & 0 \\ 0 & 0 & W_1 & W_2 \\ W_2 & 0 & 0 & W_1 \end{pmatrix}, W(2) = \begin{pmatrix} W_1 & 0 & W_2 & 0 \\ 0 & W_1 & 0 & W_2 \\ W_2 & 0 & W_1 & 0 \\ 0 & W_2 & 0 & W_1 \end{pmatrix}.$$

$W(2)$ consists of 4 blocks, each of which is a block-matrix consisting of $0, W_1$ or $0, W_2$.

Using equation (6.3) leads to the state of N within each $T_j := ({}^{(p)}\widehat{x})_j$. Computing the actual output $\mathcal{F}(T_j)$ is done by completing the transition function g to the net function $\mathcal{F}_N$. For the Elman net with linear output layer this is done by applying the hidden-to-output weight matrix: $\mathcal{F}(T_j) = O \cdot y_j^D$.

Under the given preconditions the mapping $y_*^0 \mapsto y_*^D$ can be injective, if certain requirements hold for W_1 and W_2, as shown in the next section.

6.6.2.2 Invertibility

For a recursive net that has been activated on a binary tree it is not possible to uniquely specify the states of the roots children that must have led to the certain state in the root itself. This is caused by $2|\mathcal{K}|$ values that are mapped to $|\mathcal{K}|$ values. Example: consider an Elman net with regular weight matrices W_1, W_2 is given and $s_1, s_2 \in \mathbb{R}^h$ are states of childtrees such that using the transition function g holds $s_0 = g(0; s_1; s_2)$ (with s_0 being the state in the root). Then for any $x \in \mathbb{R}^h$ the same state can be acquired using $s_1' := s_1 - W_1^{-1}W_2x$ and $s_2' := s_2 + x$ because of $W_1s_1' + W_2s_2' = W_1s_1 + W_2s_2$.

But under certain conditions (fan-out $k = 2$ or sequence length $= k^D$) for the periodic mode holds that every child in the merged tree has k pairwise different parents, so that for a given sequence the whole state transition may become injective. In the following the sequence length 2^D is considered, based on the formulas and block matrices in chapter 6.6.2.1.

Linear case

In the linear case $f = \mathrm{id}$ the mapping function $F : y_*^0 \mapsto y_*^D$ is of the form $F : y_*^0 \mapsto W(D) \cdot \ldots \cdot W(1) \cdot y_*^0$ with matrices $A := diag(W_1, \ldots, W_1)$, $B := diag(W_2, \ldots, W_2)$ and

$$W = \begin{pmatrix} A & B & 0 & \cdots & 0 \\ & \ddots & \ddots & & \\ 0 & \cdots & 0 & A & B \\ B & 0 & \cdots & 0 & A \end{pmatrix}.$$

Each block matrix $W(d)$ can be symbolically inverted using the Gauss-Jordan-algorithm. The abbreviation $n = n(d) := 2^{D-d+1}$ for the amount of block-rows and the following matrices will be used:

$$\begin{aligned} P = P(d) &:= -A^{-1}B, \\ Q = Q(d) &:= -BA^{-1}, \\ R = R(d) &:= Q^{n-1}B + A. \end{aligned}$$

It is $R = Q^{n-1}B + A = BP^{n-1} + A$ because of

$$BP^i = B(-A^{-1}B)(-A^{-1}B)^{i-1} = (-BA^{-1})(-BA^{-1})^{i-1}B = Q^iB. \tag{6.4}$$

The inverse of W can be composed as a block matrix over block-row index i and block-column index j by (powers of) the above abbreviations. It is $W^{-1} = V_1 + V_2$ with

$$V_1 = \begin{pmatrix} P^{n-1}R^{-1}Q^1 & \cdots & P^{n-1}R^{-1}Q^j & \cdots & P^{n-1}R^{-1}Q^{n-1} & P^{n-1}R^{-1}Q^0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ P^{n-i}R^{-1}Q^1 & \cdots & P^{n-i}R^{-1}Q^j & \cdots & P^{n-i}R^{-1}Q^{n-1} & P^{n-i}R^{-1}Q^0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ P^0R^{-1}Q^1 & \cdots & P^0R^{-1}Q^j & \cdots & P^0R^{-1}Q^{n-1} & P^0R^{-1}Q^0 \end{pmatrix}$$

$$V_2 = \begin{pmatrix} P^0A^{-1} & \cdots & P^{i-1}A^{-1} & \cdots & P^{n-2}A^{-1} & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & P^0A^{-1} & \cdots & P^{n-i-1}A^{-1} & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & \cdots & P^0A^{-1} & 0 \\ 0 & \cdots & 0 & \cdots & 0 & 0 \end{pmatrix}$$

where $P^0 = Q^0 = E$ is the unity matrix. That $W \cdot (V_1 + V_2) = E$ truly holds can be seen as follows:

i) It is $AP = -B$ which leads to $WV_2 = diag(E, \ldots, E, 0) + V_3$ with V_3 being zero except for the last block-row because for each but the last row of W the summands cancel out each other. The last block-row of V_3 is

$$\left(BP^0A^{-1},\ \ldots,\ BP^{n-2}A^{-1},\ 0\right).$$

Using equation (6.4) the last row equals

$$\left(BA^{-1},\ \ldots,\ Q^{n-2}BA^{-1},\ 0\right) = \left(-Q^1,\ \ldots, -Q^{n-1},\ 0\right).$$

ii) Within WV_1 all summands cancel each other out except for the last row which is of the form

$$\left((BP^{n-1} + A)R^{-1}Q^1, \ldots, (BP^{n-1} + A)R^{-1}Q^{n-1}, (BP^{n-1} + A)R^{-1}\right)$$

which can be written as

$$\left((Q^{n-1}B + A)R^{-1}Q^1, \ldots, (Q^{n-1}B + A)R^{-1}Q^{n-1}, (Q^{n-1}B + A)R^{-1}\right)$$

and by applying the definition of R this equals

$$\left(Q^1,\ \ldots,\ Q^{n-1},\ E\right).$$

iii) The two rows cancel out each other except for the last element which is E and this results in $W(V_1 + V_2) = diag(E, \ldots, E)$ yielding $W^{-1} = V_1 + V_2$.

The existence of this matrix depends on the existence of A^{-1} and $R^{-1} = (A + (-BA^{-1})^n)^{-1}$ which are block matrices. Therefore the existence reduces to the invertibility of W_1 and $W_1 + (-W_2 W_1^{-1})^n$ for $n = 2^{D-d+1}$ with $1 \leq d \leq D-1$. This could be ensured by enforcing weight matrices with sufficiently large eigenvalues for W_1 respectively small eigenvalues for W_2. It is a notable fact that for a triangular matrix the eigenvalues are explicit at the diagonal.

If the invertibility of each $W(d)$ is ensured, the reverse state transition $F^{-1} : y_*^D \mapsto y_*^0$ can be computed via $W(1)^{-1} \ldots W(D)^{-1} y_*^D$.

Ambiguity occurs only in the mapping of the state of the last layer to the actual output neurons (for example by inverting a mapping $\mathbb{R}^h \to \mathbb{R}$) and the mapping of the input labels to the first state layer (which can lead to contradictions). Numerically this problem can be approached by computing $M^* := \Pi_{d=1}^D M(d)$ which is a matrix with blocks M_{ij}^*. For a net with input and output size 1 the scalar expressions $m_{ij} := O \cdot M_{ij}^* \cdot I$ can be formed and the net function is then $\mathcal{F}_N({}^{(p)}\widehat{x}) = (m_{ij})_{ij} \cdot x$. By numerically computing the (pseudo) inverse of $(m_{ij})_{ij}$ preimages of a given class $y(x)$ and values in an open neighbourhood $y(x) \pm \epsilon$ can be computed.

Non-linear case in neighbourhood of a fixed point

In the case that no linear activation function is used, invertibility requires each state $W^{-1} y_*^d$ to be within the range of the activation function f such that f^{-1} can be applied. If however the states y_i^D are close to a fixed point of the transition function, the overall state transition still behaves linear:
Consider $f = \tanh$ for an Elman net. It is $\tanh(0) = 0$ and $\tanh'(0) = 1$. Let g be the mapping function for one specific layer in (6.3). Then the total derivative of g at a fixed point $\xi_* := \vec{0}; \ldots; \vec{0}$ is determined only by the weight matrix W: $dg|_\xi = W$. Using the Taylor formula

$$\exists 0 \leq t \leq 1 : g(\xi + h) = g(\xi) + J_g|_\xi \cdot h + h^\top \cdot H_g|_{\xi_* + t(h - \xi_*)} \cdot h$$

with $J_g|_{\xi_*} = dg|_{\xi_*}$ as the Jacobian matrix and H_g as the Hessian matrix, the net function can thus be approximated as $g(\xi_* + h) = \xi_* + Wh = Wh$ within a sufficiently

small neighbourhood of ξ_*. That means, if the previous state was close to a fixed point, the state transition equals a linear mapping.

6.6.2.3 Comparison to the Fast Fourier Transformation

A Fourier Transformation is the mapping of a periodic function to the coefficients of an interpolating polynomial. To achieve this, the function is assumed to have its values on the complex unit circle and the interpolation is performed at a finite set of points distributed over this circle. Calculating the coefficients of the interpolating polynomial is in general achieved by inverting a matrix. If, however, the number of points is a power of 2 and are distributed equally spaced, the computation of the coefficients can be done using a recursive formula called Fast Fourier Transformation (FFT) that is implemented for example by the Sande and Tukey algorithm. Details can be found in [SB07]. Without the need for a detailed explanation, the formula has an obvious recursive form when dealing with 2^n points:

$$\begin{aligned} f_{r,k}^{(m-1)} &= f_{r,k}^{(m)} + f_{r,k+2^{m-1}}^{(m)}, \\ f_{r+2^{n-m},k}^{(m-1)} &= (f_{r,k}^{(m)} - f_{r,k+2^{m-1}}^{(m)})\epsilon_m^k. \end{aligned}$$

The formula is computed for $m = n$ based on initial values for $f^{(n)}$, which are the function values itself, ranging down to $m = 0$. For a fixed $m - 1$ the other indices range at $0 \leq r \leq 2^{n-m}$ and $0 \leq k \leq 2^{m-1}$. Within one "layer" of indices (r, k) at level $m - 1$, a linear combination of entries in the previous layer m is formed where ϵ_m is a complex, scalar value. The linear combination however does not only use local indices (r, k) and $(r + s_1, k + s_2)$ with fixed $s_1, s_2 \in \mathbb{Z}$ to access "previous" or "future" parts of a certain sequence. It rather includes entries that are shifted with a stride 2^{m-1}, that is, an exponentially varying stride.

The computational capability of the FFT lies in the nifty reorganisation of the input data by accessing arrays at indices with exponentially varying stride.

While the periodic mode does of course not resemble the same recursion formula as the FFT, its mere existence makes the assumption plausible that assuming linearity (for example by using $f = \mathrm{id}$ as activation function), does not automatically render the periodic mode to have no computational power.

In the non-linear case, if every component of $W(d)^{-1}y$ (with y being the intermediate states) is within range of the activation function, the state transition is still invertible. If $W(d)$ is not regular, then vectors x with $W(d)x = y$ can be searched so that x is within the range of the activation function. This will probably not be feasible as the following example shows for a very simple net.

Preimages of net output

Let the parameters of the net seen in figure 6.8 be $x = (x_1, x_2)^\top$, $y = (y_1, y_2)^\top$, $W = \left(\begin{smallmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{smallmatrix}\right)$, $a = (a_1, a_2)^\top$, $a_1 \neq 0 \neq a_2$. With $y = \mathrm{sgd}(W^\top x)$, $z = \mathrm{sgd}(a^\top y)$ is

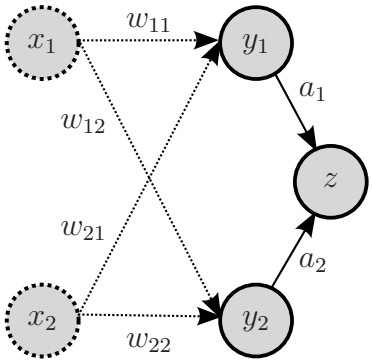

Figure 6.8. *Net for which preimages of an output value z are searched.*

the output of the net with $z \in (0, 1)$ and let $\tilde{z} := \mathrm{sgd}^{-1}(z) = a^\top y$. Finding possible activations y that result in the given output means to solve the previous linear equation system:

$$y = \begin{pmatrix} \tilde{z}/a_1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} -a_2/a_1 \\ 1 \end{pmatrix}, \mu \in \mathbb{R}.$$

For any μ this produces a valid solution to the equation system. However, it is $y = \mathrm{sgd}(W^\top x)$ and $\mathrm{sgd}(\mathbb{R}) = (0, 1)$. Hence, y is considered a *plausible preimage* if and only if $0 < y_1, y_2 < 1$. Here this is equal to

$$\begin{cases} 0 < \tilde{z}/a_1 - \mu a_2/a_1 < 1 & (I) \\ 0 < \mu < 1 & (II) \end{cases} \text{with } (I) \Leftrightarrow \begin{cases} \tilde{z}/a_2 < \mu < (\tilde{z} - a_1)/a_2 & (a_1/a_2 < 0) \\ (\tilde{z} - a_1)/a_2 < \mu < \tilde{z}/a_2 & (a_1/a_2 > 0) \end{cases}$$

and using $\alpha := (\tilde{z} - a_1)/a_2, \beta := \tilde{z}/a_2$ it is $(I) \Leftrightarrow \mu \in (\min\{\alpha, \beta\}, \max\{\alpha, \beta\})$.
So there is no solution if $\max \alpha, \beta \leq 0$ or $\min \alpha, \beta \geq 1$ because $\tilde{z}$ ranges in $(0, 1)$. By analysing each case, the following tabular is acquired, stating when there is *no plausible solution* for $\tilde{z} = a^\top y$:

	$a_1 < 0$	$a_1 > 0$
$a_2 < 0$	$\tilde{z} > 0$	$\tilde{z} > a_1$
$a_2 > 0$	$\tilde{z} > a_2$	$\tilde{z} > a_1 + a_2$

So if at least one weight is ≥ 1, always the whole interval $(0, 1)$ is plausible and if all weights are ≤ 0, there is no plausible solution even though the equation system still has an infinite number of solutions. Searching further preimages could be done for arbitrarily chosen values, for example by partitioning the plausible interval (y_{lower}, y_{upper}) into n smaller intervals and picking a representative from each part.

The previous consideration holds for any constellation where two neurons serve as input for one. This could be the case when they are the result of an unfolding process for a tree t, for example when $z \succcurlyeq z$ without any other (including feedforward) connections: having $v_{z,z} = (a_1, a_2)^\top$, this case can be transferred as $z = z(t)$, $y_1 = z(\chi_1(t))$, $y_2 = z(\chi_2(t))$. In this case one could additionally assume that feedforward connections from an input layer to z exist but the input data is 0.

The case is the same for a self-recurrent neuron with one connection from one input neuron $z \succcurlyeq z, i \to z$ with $k = 1$, $v_{zz} = a_2$ and $w_{iz} = a_1$ after unfolding for a sequence: this case can be transferred by $z = z(t)$, $y_2 = z(t-1)$, $y_1 = i(t)$.

When two neurons serve as input for two neurons by $y = \mathrm{sgd}(W^\top x)$ plausible solutions for $W^\top x = \tilde{y}$ are searched, there can either be no solutions because the system is not solvable, there can be a unique solution that can be plausible or not and again there can be an infinite number of solutions.

6.6.3 Divide and Conquer

The algorithm for applying a recursive net on ${}^{(m)}\widehat{x}$ is very easy to implement in the recursive form in algorithm 4. It implements a function named "DC" that takes three arguments: the sequence x, the beginning L and the end R of the indices to consider. The other necessary variables (transition function f of a recursive net N with initial context ξ and fan-out $k = 2$, rounding function r) are assumed to be globally available.

Algorithm 4 DC(x, L, R)

Require: $x \in \left(\mathbb{R}^l\right)^n$, left limit L, right limit R
Ensure: $DC(x, 1, n)$ is the state of N within ${}^{(m)}\widehat{x}$
1: $c_1 \leftarrow \xi$
2: $c_2 \leftarrow \xi$
3: $M \leftarrow r\left((L+R)/2\right)$
4: **if** $L < M$ **then**
5: $\quad c_1 \leftarrow DC(x, L, M-1)$
6: **end if**
7: **if** $R > M$ **then**
8: $\quad c_2 \leftarrow DC(x, M+1, R)$
9: **end if**
10: **return** $f(x_M; c_1; c_2)$

The Bidirectional Divide and Conquer can be implemented via algorithm 5 without temporarily creating the structure ${}^{(m)}\widehat{x} * {}^{(m)}\widehat{x}$ by keeping track of the root-path that is attached to the 3rd position.

Algorithm 5 BDDC(x, L, R, y)

Require: $x \in \left(\mathbb{R}^l\right)^n$, left limit L, right limit R, state y
Ensure: $BDDC(x, 1, n, \xi)$ is the state of N within ${}^{(m)}\widehat{x} * {}^{(m)}\widehat{x}$
1: $c_1 \leftarrow \xi$
2: $c_2 \leftarrow \xi$
3: $M \leftarrow r\left((L+R)/2\right)$
4: rpfc$\leftarrow f(x_M; \xi; \xi; y)$ ▷ state of N in the root-path including this node
5: **if** $L < M$ **then**
6: $\quad c_1 \leftarrow BDDC(x, L, M-1, rpfc)$
7: **end if**
8: **if** $R > M$ **then**
9: $\quad c_2 \leftarrow BDDC(x, M+1, R, rpfc)$
10: **end if**
11: **return** $f(x_M; c_1; c_2; y)$

6.7 Summary

Several methods for mapping sequences to structures have been introduced in this chapter. Algorithms have been proposed for computing the output of a recursive net when processing such a structure. The concept behind bidirectional nets has been generalised onto trees. It has been adapted to be suitable for classification tasks and has been combined with one of the novel methods. What remains to be done is to compare those methods on actual problems of machine learning to determine if they are practically suited in general or at least for specific tasks in order to achieve lower error rates and better generalisation capability.

7 Related Work

Neural networks are usually adapted to the problem they are supposed to deal with. If chemical structural formulas are part of this problem, then recursive nets are used. If sequential data is dealt with, then recurrent nets are used. This chapter summarises some articles describing methods that can relate to the matter of creating (tree-)structures from sequences or two-dimensional manifolds. They relate to this thesis in the sense that the architectural bias of *recursive* nets on the same, sequential data can now be examined (7.3). Others use and generalise the concept of bidirectional processing by applying it on network architectures different from Elman nets or by applying them for two-dimensional inputs (7.1). Because the restructuring methods described in this thesis put every element of a given sequence into another context, contextual architectures relate to this thesis as well (7.2). Finally, methods are described that analyse the content of sequential (one- or two-dimensional) data to form trees on which a recursive net can be trained (7.4). None of the mentioned papers however explicitly specify a mapping function from the space of sequences into the space of trees. Some articles deal with a so-called architectural bias of recurrent nets. An explicit process of restructuring as a step of preprocessing is however not mentioned in the listed papers.

7.1 Bidirectional architectures

The first paper to mention is of course [SP97]: “Bidirectional Recurrent Neural Networks” from Mike Schuster and Kuldip K. Paliwal in 1997, as already mentioned in chapter 4.5 and 6.2.2.

The architecture has been used together with MLPs in [Bal+01]: “Bidirectional Dynamics for Protein Secondary Structure Prediction” by Baldi, Brunak, Frasconi, Pollastri and Soda in 2001. Therein methods for predicting the physical shape of proteins basing on their genetic encoding have been examined.

In [BP03]: “The Principled Design of Large-Scale Recursive Neural Network Architectures–DAG-RNNs and the Protein Structure Prediction Problem” by Baldi and Pollastri in 2003 these methods have been generalised for two-dimensional

(and "D-Dimensional") data. They mention to use recursive nets ("DAG-RNN") which may be due to the shape of the depicted nets that have been unfolded into feedforward nets for explanation.

The LSTM architecture has been combined with the bidirectional idea in [GS05]: "Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures" by Graves and Schmidhuber in 2005 to conduct speech recognition.

7.2 Contextual architectures

The notion of "context windows" is mentioned in [MSS04]: "Contextual Processing of Structured Data by Recursive Cascade Correlation" by Micheli, Sona and Sperduti in 2004. Within this article, the shift operators q^{-1} and q^{+1} are defined and combined for a generalised cascaded network. The Contextual Cascade Correlation is a process that develops the network architecture (i.e. the size of the net) during training. Not only activations from child nodes are used for hidden layer activation, but also from parent nodes from neurons that could already be activated due to the nature of the cascade architecture (compare chapter 4.7). Applied to a sequence, for example, this enables the architecture to access computations from future position $n + 1$ during activation at position n.

7.3 The architectural bias

"The Applicability of Recurrent Neural Networks for Biological Sequence Analysis" has been examined in [HB05] by Hawkins and Boden in 2005. The work is motivated by research dealing with the fact that the recurrent neural network architecture itself, without prior training of certain weights, is already suited for dealing with the classification of sequential data. This roughly means that during activating a random net the states of the untrained net correlate to the desired target to a certain degree.

A trivial way of saying that recursive nets have the same architectural bias is to state that by $k = 1$ the recurrent net is a special case of recursive nets. Through introducing restructuring algorithms to map sequences to trees the question arises to what extend recursive nets non-trivially have an architectural bias towards classifying *sequences*. This question can be raised for an arbitrary fan-out k using recursive and periodic restructuring and with $k = 2$ ($k = 3$) using (bidirectional) Divide and Conquer.

7.4 Graph based and graph creation

Recursive nets have been generalised to operate on graphs that have no ordering of the input nodes, but labels connected to the directed edges between the nodes. [Bia+05]: "Recursive neural networks for processing graphs with labelled edges: theory and applications" by Bianchini, Maggini, Sarti and Scarselli in 2005 proposes such an architecture for labels that are vectors in $\mathbb{R}^d$. This is done by using a weight tensor that has d layers. The actual recursive net input into a neuron is computed not by selecting the layer of the tensor that correlates to a childs position (as the position is not defined) but rather by a linear combination from each layer, wherein each linear factor is the corresponding component of the edges label. Obviously this generalises recursive nets, because for a fanout k each position can be assigned to a unit vector $e_\kappa, 1 \leq \kappa \leq k$. The advantage is that small deviations in a label have only small influence. These deviations could result for example by encoding parameters from a physical entity that is subject to measuring inaccuracies. The architecture is used for face recognition by preprocessing a given image into areas that are modelled as nodes of an input graph. The edge labels are certain angles derived from the areas. This method requires content related preprocessing prior to using the recursive nets.

"The Graph Neural Network Model" is introduced in [Sca+09] by Scarselli, Gori, Tsoi, Hagenbuchner and Monfardini in 2009. It defines neural networks that take undirected graphs as input data. This is achieved by defining the output as the solution of a system of implicit functions.

An algorithm that also deduces structure from sequential, one-dimensional and two-dimensional input is described in [Soc+11]: "Parsing Natural Scenes and Natural Language with Recursive Neural Networks" by Socher, Lin, Ng, Manning in 2011. The algorithm does not base on preprocessing the input data for sentences, for pictures however several preprocessing methods are used to reduce the (structural) size of the input image. Afterwards the algorithm operates on the "set of all possible trees" that can be constructed out of the input. As a result, structures are created that are supposed to reflect the given input. For sentences this is the grammatical structure, for pictures the structure is the "scene" of the image.

8 Experimental evaluation of the proposed methods

In this chapter several experiments and their outcome will be described. At first general information about the settings of these experiments are given and the general sort of pattern data is grouped and the notations used to describe each individual setup are clarified. A brief explanation of the program that was developed to execute the learning task is given. After explaining the order in which the experiments have been conducted, the general layout of a report over an experiment is described and one report per experiment is given per section. The chapter concludes with an overall summary on how the different restructuring methods perform in comparison to the recurrent default processing.

8.1 Settings and general information

Sorts of input data

Each experiment uses a certain pattern set that has specific properties: it contains either discrete valued (symbolic) data or continuous, real-valued data. It can be synthetic data, generated by a well-known function that is to be approximated (in numerical literature also referred to as "toy problem" - not implying that it would be easy to approach). Or it can be so-called "real world data" consisting of data usually gathered from sensor readings or other sources from the real world (genome sequences for example that are attributed into families and superfamilies).

Data consisting of discrete values is a sequence of arbitrary length consisting of vectors of a fixed length $l \in \mathbb{N}$. Each vector usually contains exactly one value 1 at the position i and 0 (or another fixed value, for example -1) on all other positions. By this, the 1 at position i encodes the unit vector $e_i \in \mathbb{R}^l$ which represents a specific symbol out of a set of l pairwise different symbols. The vector $(0, 1, 0)$ for example represents the second out of three symbols.

When the data is real-valued (continuous), the vectors still have a fixed length l but their elements contain arbitrary values that are usually (pointwise) normalised into an interval $[-1, 1]$ or $[0, 1]$.

Notations

A LSTM neural net of size n consists of n blocks of size 1 as described in 5.1. An Elman net of size n consists of n hidden neurons as described in 4.4. Both nets input and output layers sizes are adopted to the size of the input data: in the following experiments, the output layer always has size 1 because single-class classification problems are dealt with; the size of the input layer ranges between 1 and 26, depending on the dimension l of the input space $(\mathbb{R}^l)^*$ and whether or not a terminator symbol is attached (see below). Within the reports, the following abbreviations will be used:

string	**meaning**	**string**	**meaning**
ms	sequential mode	size	the size of the net
mrc2	recursive mode, $k = 2$	mrc2T	modes with dynamically attached terminator symbol
mrcysdp2	periodic mode, $k = 2$	mrcysdp2T	
mrm	Divide and Conquer	mrmT	
mrmbd	bidirectional D&C	mrmbdT	
reg	Elman net	lstm	LSTM net
ERR	error rate in unit [1]	MSE	mean squared error
train.	ERR on training set in [%]	gen.	ERR on testing set in[%]

The sequential mode is the processing of the sequence in the usual way, that is, with a recurrent network. The abbreviations for the restructuring modes actually result from the command line parameters issued to the program that was developed to, amongst others, conduct all the experiments described in this chapter. The error rate is the fraction "correctly classified patterns divide by number of patterns".

The terminator symbol

A very simple method of formally preprocessing a given sequence is to attach a starting and ending symbol by increasing the input size by 2 and using unary encoding for these symbols. This way, the neural net can symbolically see when the sequence starts and ends. This procedure has been used in the experiments on LSTM described in [HS97]. Within the experiments in this thesis, attaching a terminator symbol, that is, a unique symbol that does not occur in the input sequence, has been done for some experiments. Using a terminator symbol can be understood as a formal preprocessing, as it does not depend on analysing the given input. It however is not a restructuring method, because the input sequence keeps

its form: its length (and each individuals vectors size) is just increased by 1. A sequence $(x_1, \ldots, x_n)$ is thereby mapped to a sequence $(x_1; 0, \ldots, x_n; 0, \vec{0}; 1)$.

Program

The program is written in C++ and implements the training algorithm RPROP$^-$ with gradient initialisation as described in 3.4. For LSTM, the formulas from 5.2 are implemented for updating the error signals and computing the gradient. For Elman nets full RTRL with optimisation towards (i.e. ignoring update signals for) Omega neurons is used to reduce unnecessary computations as described in 3.2.4. All used nets are biased and the Elman net uses the sigmoidal activation function. The initial context is not learned and set to 0.

Input data is stored as sparse vectors using the "Serialization" library from BOOST[1]. Also the libraries "Thread" and "Program Options" are used to compute one pattern per CPU (as the program has been executed on a computer with 4 cores) and to set configurations (input data, mode, weight span etc.) as dynamic as possible.

Though all experiments that are described in the following are run with RPROP$^-$ on a static training set, the program is capable of doing parallel online learning using gradient descent with and without momentum term by gathering the patterns from a library that is dynamically loaded via command line parameters. This is due to the fact that the implementation especially of LSTM was being tested by resembling some experiments described in [HS97], all of which are conducted online, that is, by iteratively creating new (synthetic) training patterns.

As training requires a net with random weights to start with, the "mt19937" implementation of the Mersenne Twister from the "Random" library from BOOST has been used as random number generator; it has a cycle length of $2^{19937} - 1$ numbers with 32 bits each. Randomly generated patterns (as for the UDXOR) are generated from their own instance of this random number generator. As of course the numbers are pseudo-random, different rounds of the same experiment are created by randomising the weight matrices multiple times, taking new pseudo-random numbers from the generator. Doing so allowed to repeat the same round (after fixing a bug, for example) with the exact same, pseudo-random weight matrix without the need of storing it. The amount of randomisations determines the "round" of the experiment. The initial weight span is just a factor for each weight so that a weights value w from round N that originates from an initial weight span of w_0 will have the value $2w$ in round N if the initial weight span $2w_0$ is used.

[1] `www.boost.org`

For computing the sigmoidal activation function, the exponential approximation as described in [Sch99] has been used.

The data sets

The following data sets have been used as training patterns to examine the different restructuring modes:

1. Synthetically generated, symbolic data of dynamic length ("UDXOR"). On this data set the most experiments regarding the different training parameters (network size, network kind, pattern length) have been conducted to gather an overview on how the different modes perform.

2. Real-world, real-valued data of huge but fixed length ("ARCENE") taken from a contest on machine learning (feature extraction). This data set will show that the initial weight span does have an influence. The content of the data had been preprocessed for the contest to make it harder to classify.

3. Real-world, symbolic data of dynamic length ("SCOP"). This contains genome sequences of strongly varying length that are to be classified for belonging into a certain super family. In this data set the positive classes are much fewer in number than the negative classes, so a special training method has been used to account for this. The nets have been re-trained by initialising each setup with the final result of the non-specialised training, so that the gradient initialisation for RPROP$^-$ could show its impact.

4. Real-world, real-valued data of very huge but fixed length ("NCIOvarian"). This data set has been created basing on the idea of the ARCENE data set but preprocessing has been avoided. This is also the only experiment where not only the performance in learning, but also the performance in generalisation has been focused on from the beginning.

All data sets are supposed to contain long sequences, so that long term dependencies can be expected. Their primary focus is to check whether learning with the proposed methods is possible at all. Many experiments performed that good that excessive overfitting occurred, sadly rendering many experiments not well suited for comparing the quality of generalisation. For all experiments, the maximum absolute value for a single weight was not limited and for comparing all experiments in equal many have not been stopped by defining some stop criteria. Because of this, the NCIOvarian data set has been created and the experiments have been conducted with a stop criteria that leaves the error rate on the training set high enough to have convincing generalisation errors.

Layout of the reports

The following reports all have the same structure and order, containing:

- the nature of the data set underlying the experiment,
- a graphical display of an example pattern,
- the specific settings used to learn on the pattern set,
- plots of the iterative learning of selected modes arranged for comparison and
- comments on some specific behaviour,
- tables summarising the error rates at the end of the training and showing the error rates at the test set and
- a summary and conclusion about the performance of each mode.

Plots of mean squared error are shown even if the training was conducted with cross entropy based error function as it correlates closer to the error rate. Above each specific plot the kind of the net, the mode and the average training and generalisation error are given; the error rates can of course also be found in the tables. Each and every error rate in the following plots and tables is the average over several rounds (9 or 10 in amount as described).

8.2 The "UDXOR" data set

This data set resembles a strongly delayed exclusive OR function with symbols in unary encoding ("unary delayed XOR"). Usage of this pattern set is motivated by [HS97] where it has been reported that LSTM has been found unable to solve problems of the kind "strong delayed XOR". This experiment is to find out about *how far* problems of this kind can be solved with an LSTM net.

Subject is to learn the mapping $F : P \to C$ with pattern space P of pattern $p = (p_i)_{i=1}^{n}$ and $C = \{0, 1\}$. A symbol set $S = \{e_i | 1 \leq i \leq 6\}$ with unit vectors $e_i = (\delta_{n,i})_{n=1}^{6}$ is defined. The pattern basis length $n_{min} \in \{100, 200, 300\}$ (concrete choice depending on the setup) is randomly enlarged by up to 20% (equally distributed) for an actual pattern. Positions of relevant symbols within each pattern are fixed at $i_1 = \lfloor n/3 \rfloor$ ($33 \ldots 40$ for $n_{min} = 100$, for example) and $i_2 = \lfloor 2n/3 \rfloor$ ($66 \ldots 80$). Each pattern is a sequence of randomly chosen symbols such that holds:

$$p_i \in \begin{cases} \{e_1, e_2, e_3, e_4\} & \text{if } i_1 \neq i \neq i_2, \\ \{e_5, e_6\} & \text{if } i = i_1 \vee i = i_2. \end{cases}$$

The classification target is determined by the logical value of the statement:
"Either e_5 is present, or e_6 is present"
The symbols e_1 to e_4 are therefore only to disturb the training process. Considering $e_{5,6}$ to represent the logical values 'true' and 'false', this statement represents the logical exclusive-OR.

Considering variable positions for i_1 and i_2 for the set of all possible patterns, the pattern set is translation invariant because the position of the symbols e_5 and e_6 are not important for the target class. Therefore, according to section 6.5 the periodic mode should perform very good in comparison to the recursive mode, because the amount of trainable pattern is drastically increased by a factor > 100, that is, the length of the sequences, with computational costs only increasing by factor $\log_2 100 \approx 6.7$.

$$\begin{aligned} \text{AAAAAA}\mathbf{E}\text{BBBBBB}\mathbf{E}\text{CCCCC} &\mapsto 0 \\ \text{AAA}\mathbf{E}\text{ABBBCC}\mathbf{F}\text{DCC} &\mapsto 1 \\ \text{ABDACB}\mathbf{F}\text{DBACAB}\mathbf{E}\text{CCCDCCDC} &\mapsto 1 \\ \text{ABCDAB}\mathbf{F}\text{CDABCD}\mathbf{F}\text{ABCDAB} &\mapsto 0 \end{aligned}$$

Figure 8.1. *Manually created example patterns for the UDXOR pattern set with their respective class mappings: distortion symbols are A, B, C, D and significant symbols (emphasised in bold) are E and F.*

Settings

A LSTM net and an Elman net have been trained using $2\,\mathrm{sgd}(x) - 1$ as hidden layer activation. Each pattern set consists of 100 patterns of minimum lengths 100, 200 and 300, respectively. The validation set for each setup consists of 1000 patterns. Each of those setups has been trained using the four modes sequential, recursive, periodic and D&C. The size of each net varied between 1 and 5. This results in a total number of $2 \cdot 3 \cdot 4 \cdot 5 = 120$ different setups. For the length 100 only, these setups have been rerun with an additional terminator symbol. The initial weight span for each setups was 0.1 (i.e. small), so that all weights ranged between -0.1 and $+0.1$ before the first iteration. The iteration limit was set to 1000, so that exactly 1000 weight-updates have been committed by RPROP$^-$ before the learning ended. Each setup has been run nine times[2] (rounds) to account for a statistical somewhat representative experiment. This amount is still is small enough to plot the error rates of each individual round into the same graph. Cross entropy has been used as error function, the plots however contain the mean squared error. For LSTM, the input gates have been biased to -3 for the first and stacking -0.1 for each additional block. RPROP$^-$ has been set to start with a learning rate of 0.1

[2] the original number of 10 was reduced by one because of a technical problem

for the gradient initialisation, factors 0.5 for dropping and 1.2 for increasing the steps, 0 for minimum absolute increment (i.e. no lower limit) and 1.0 for maximum absolute increment.

The early experiment

In a previous experiment that will not be explained in deep, a variation of this setup has been conducted with a wide variety of initial weight spans (ranging from 0.1 to 6.0 in steps of 0.1) and using a "simple delayed XOR" pattern set without distortion symbols but only 0 at non-significant positions, using plain gradient descent with learning rate 0.05 (with averaged gradient over 100 freshly created pattern each) on an LSTM net with 2 blocks of size 2 and running 100 different trials for each setting. The iteration limit was set to 5 mio. to stick close to the settings in [HS97]. The training has been stopped either when a number of 5 million iterations was reached or when the average and maximum mean square error on the last 2000 training pattern was below 10^{-3} and 10^{-2} respectively. The statistics on this experiment can be found in figure 8.2 and can be interpreted in that

i) the simple delayed XOR can only be solved when using fairly large initial weight spans and

ii) only for very short pattern lengths (here: 10) a solution could be found so that training becomes more and more non-successful for increasing length.

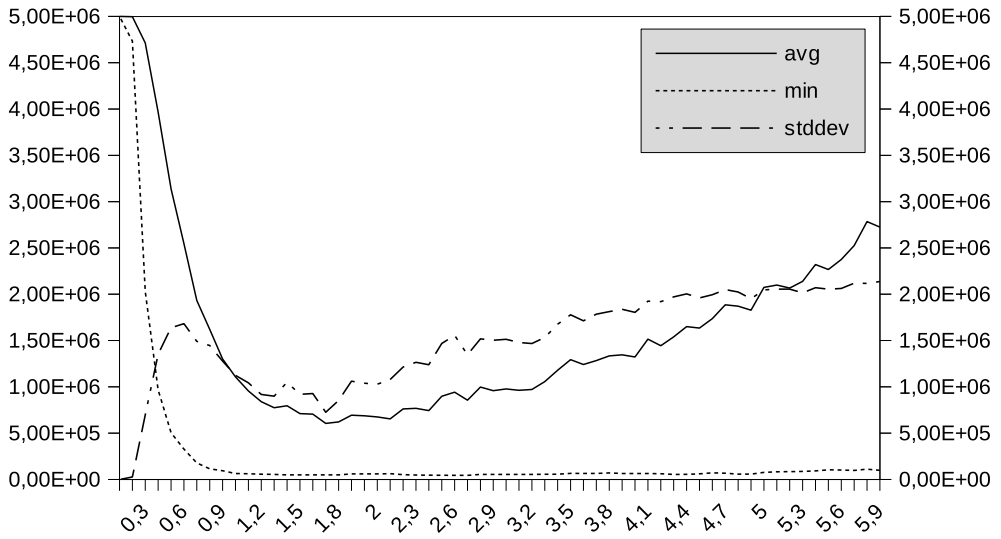

Figure 8.2. *Average, minimum and standard deviation of the average amount of iterations until stop criterion is met, plotted against the initial weight span (horizontal axis). From the early experiment on the "exclusive OR".*

This leads to the assumption that either plain gradient descent performs too bad to reach a convergence to any solution within a feasible time, or that a huge initial weight span is a necessary precondition for successful training.

The following report will show that the (tougher) UDXOR data set can be learned also with small initial weight spans, leading to the conclusion that plain gradient descent on a virtually unlimited training set might be not suited for this problem.

Fascinatingly, the report on the ARCENE data set will show that a huge weight span can still be required even when using RPROP$^-$.

8.2.1 Results

In this section, plots for selected setups will be shown. All are from nets of size 5 and from pattern sets with a base length of 100.

At first the performance of the standard sequential mode can be seen in figure 8.3: attaching a terminator symbol surprisingly worsened the results for LSTM while the Elman net was not influenced. A subset of the rounds for LSTM tend to converge towards a solution with fast and frequent changes, while the remaining rounds do not converge. The Elman net performs quite good with an average error rate of 17% and without unstable behaviour. On the validation set the Elman net produces random results, while the average for LSTM is unexpectedly good with some rounds producing less then 2% generalisation error.

The error rates of all modes with and without terminator symbol for different sizes can be found in table 8.1.

In figure 8.4 the plots for the novel restructuring modes without terminator symbol are shown. Training the LSTM nets was successful already after 200 iterations for all novel methods, while the Elman net reached fast success only for the D&C mode. The Elman net with D&C also behaves smooth over all rounds while LSTM is volatile with many spikes in the MSE. The error rates decreased in general upon attaching a terminator symbol (figure 8.5) and the state of reaching almost interpolation has been moved from < 200 iterations to around or slightly above 200 iterations.

All plots show that the training becomes unstable (i.e. an increasing MSE and error rate) a long period after reaching 0% or almost 0% training error. In practise, when the nets serve a specific task, the generalisation error would have been kept track of during the training and training would have been stopped if, for example, the generalisation error raises. Too keep the experiments as comparable as possible, they have however been ran for the whole 1000 iterations.

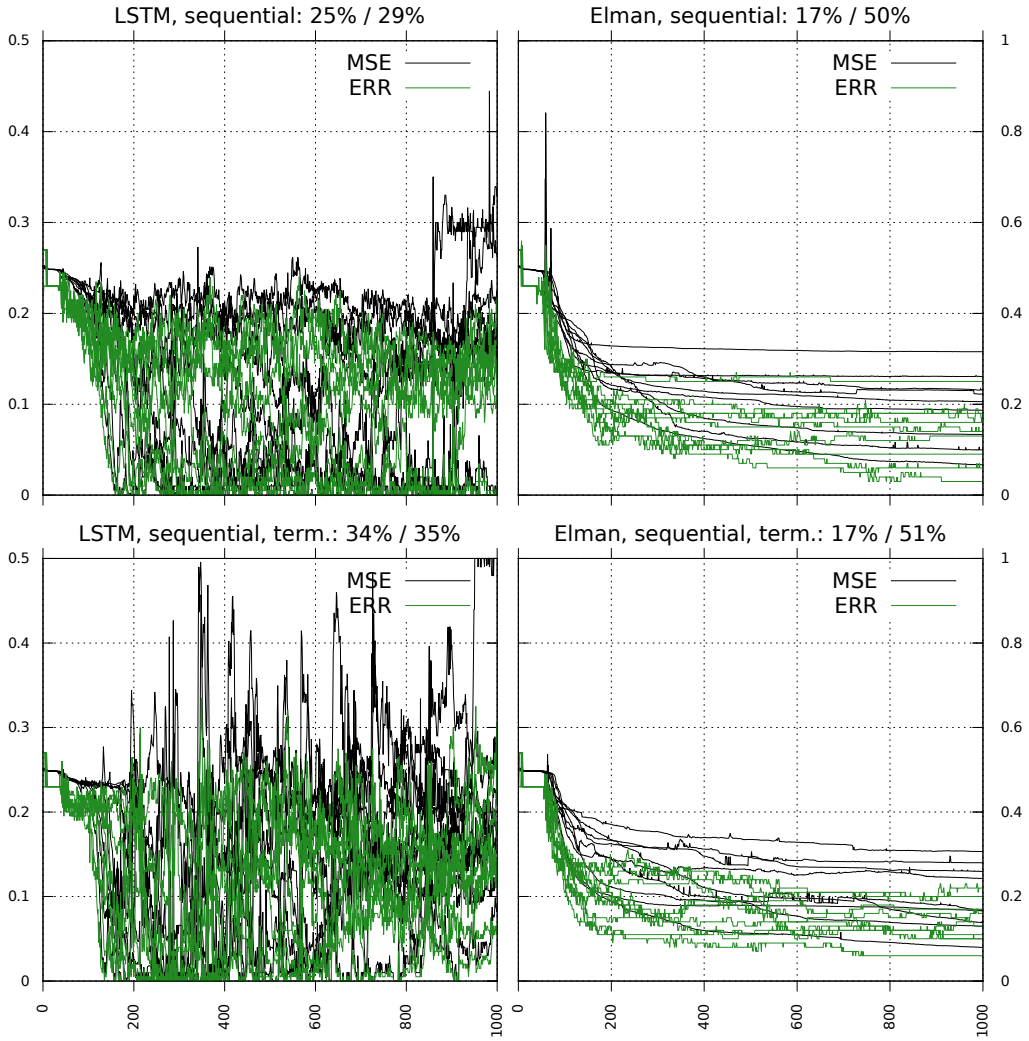

Figure 8.3. *Training graphs for sequential mode, with and without attached terminator symbol for the UDXOR data set.*

When comparing the Elman net with periodic mode from figures 8.4 and 8.5, one single successful round from the setup without terminator symbol and none with terminator symbol can be seen. This round led to 0% training error and 1% generalisation error.

The table containing training and generalisation error of all experiments on the net sizes 1, 3 and 5 can be found in table 8.2.

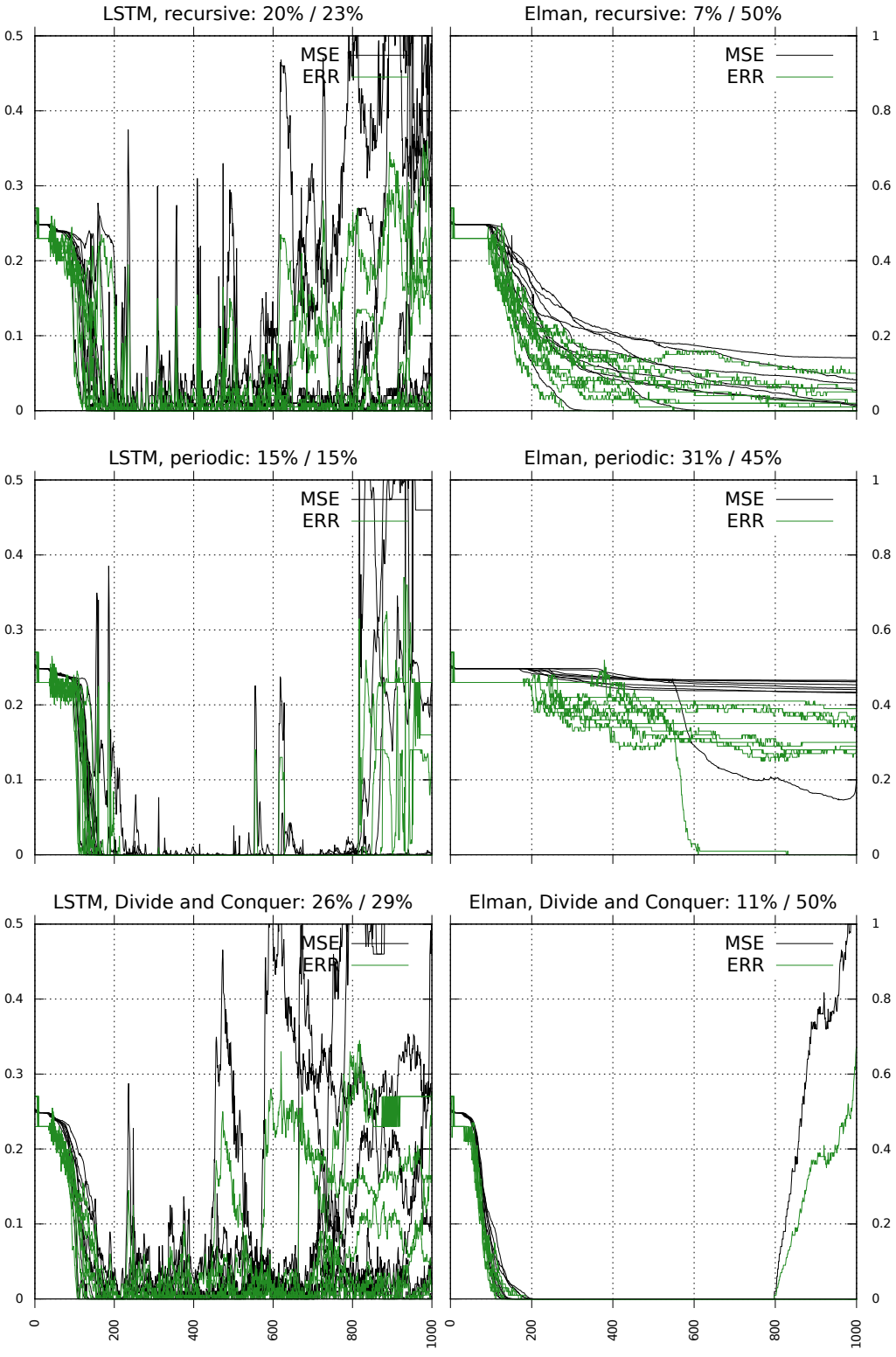

Figure 8.4. *Training graphs for the restructuring modes without terminator symbol for the UDXOR data set.*

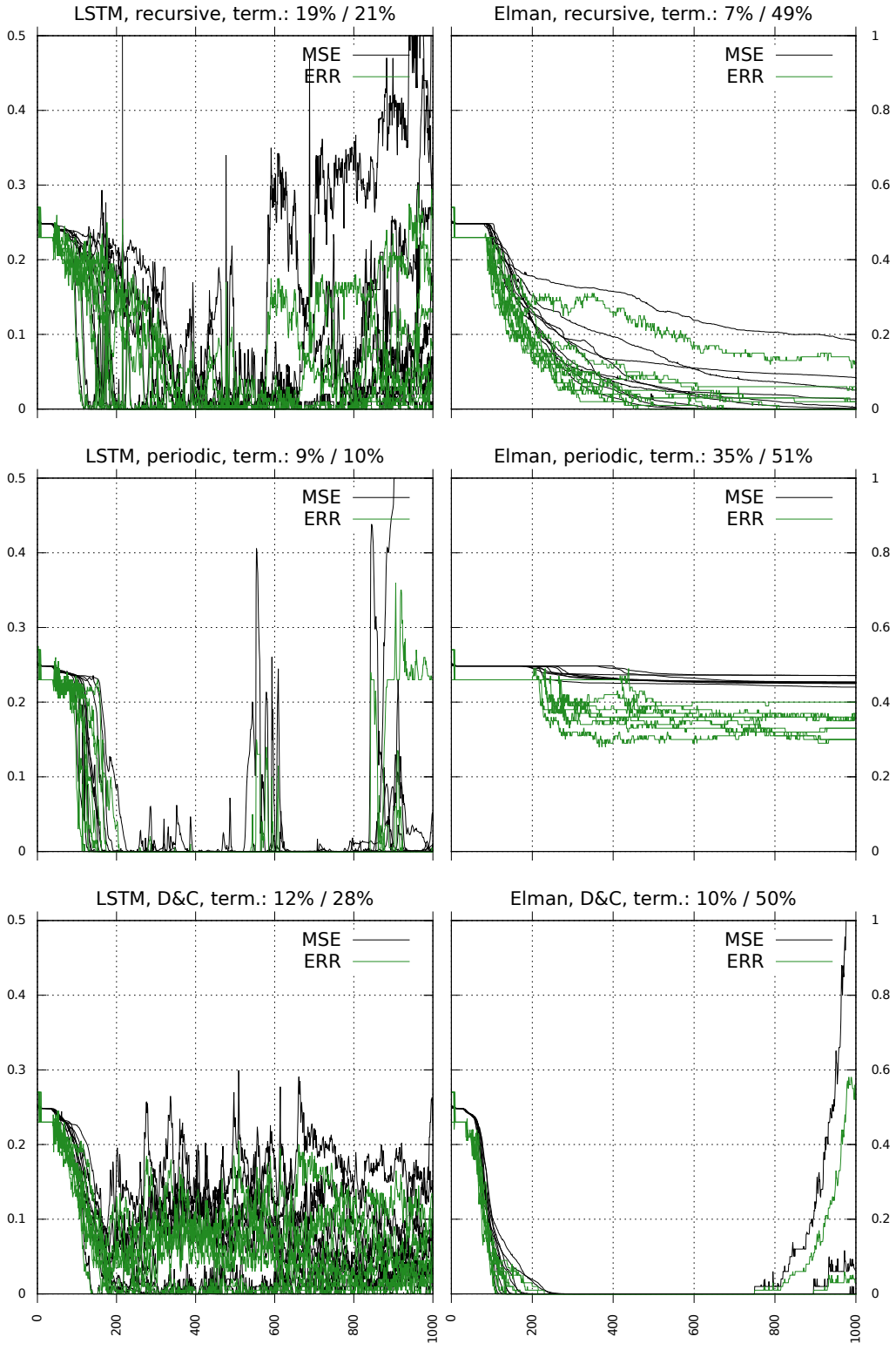

Figure 8.5. *Training graphs for the restructuring modes with attached terminator symbol for the UDXOR data set.*

net	mode	pattern	train. size 1	gen. size 1	train. size 3	gen. size 3	train. size 5	gen. size 5
lstm	mrc:2	UDXORT	8.7	10.7	9.6	13.4	18.9	21.3
lstm	mrc:2	UDXOR	10.5	13.5	24.5	24.4	20.0	22.9
lstm	mrcysdp:2	UDXORT	10.1	10.4	10.0	10.0	9.2	10.3
lstm	mrcysdp:2	UDXOR	9.2	10.3	23.0	27.0	15.1	15.3
lstm	mrm	UDXORT	33.8	50.0	7.0	17.2	11.8	27.9
lstm	mrm	UDXOR	16.9	24.6	14.7	19.9	26.0	29.3
lstm	ms	UDXORT	39.2	46.2	26.1	30.5	33.5	34.5
lstm	ms	UDXOR	39.6	48.0	25.6	32.3	24.8	28.9
reg	mrc:2	UDXORT	41.6	51.7	18.6	50.4	6.7	49.1
reg	mrc:2	UDXOR	43.7	50.6	24.6	49.3	7.0	49.8
reg	mrcysdp:2	UDXORT	45.7	51.4	39.6	50.4	35.1	50.7
reg	mrcysdp:2	UDXOR	46.0	51.3	41.4	50.1	30.6	44.9
reg	mrm	UDXORT	41.1	50.1	8.9	50.0	10.1	50.0
reg	mrm	UDXOR	33.2	50.6	8.3	50.4	11.3	49.5
reg	ms	UDXORT	37.1	50.9	23.5	50.7	17.1	51.1
reg	ms	UDXOR	33.9	51.3	24.7	51.1	16.8	50.4

Table 8.1. *Training error (ERR) and generalisation error for previously depicted training on patterns with base length 100. The pattern set named "UDXORT" represents the setting using a statically attached terminator symbol. In later experiments, the terminator symbol has been thought of belonging to the mode, instead of to the pattern set.*

8.2.2 Interpretation

The restructuring modes by using recursive networks clearly outperforms the standard sequential mode using recurrent networks, though the sequential mode using recurrent nets did not fail completely. As a conclusion, the "strong delayed exclusive-OR" can be approached when using RPROP$^-$ on a fixed training set. However, using the novel restructuring modes using a recursive net consequently results in successful training. While the recurrent nets converge to a solution in only some of the cases, the recursive nets converge straight and fast.

The disappearance of the one successful round from periodic Elman results from the order in which the weight matrices are randomised: the weights assigned outgoing from the input neuron, that contains the terminator symbol bit, would have been assigned to other, previously existing weights if the terminator symbol would not have been introduced. Thereby almost all weights values are shifted and another round is defined that is not comparable to the one without terminator symbol. This shift is repeated for each re-randomisation (round).

net	mode	length	train. size 1	gen. size 1	train. size 3	gen. size 3	train. size 5	gen. size 5
lstm	mrc:2	100	10.5	13.5	24.5	24.4	20.0	22.9
lstm	mrc:2	200	29.3	45.4	31.9	36.9	17.3	17.6
lstm	mrc:2	300	39.9	50.4	16.6	22.7	13.4	17.5
lstm	mrcysdp:2	100	9.2	10.3	23.0	27.0	15.1	15.3
lstm	mrcysdp:2	200	35.1	51.3	11.3	10.9	10.0	10.0
lstm	mrcysdp:2	300	36.7	50.3	22.0	27.6	7.0	7.5
lstm	mrm	100	16.9	24.6	14.7	19.9	26.0	29.3
lstm	mrm	200	30.1	50.2	16.7	38.2	16.2	39.1
lstm	mrm	300	29.8	50.1	19.7	32.0	13.5	20.9
lstm	ms	100	39.6	48.0	25.6	32.3	24.8	28.9
lstm	ms	200	29.9	47.5	28.0	43.8	23.2	26.6
lstm	ms	300	38.0	49.5	22.5	34.8	25.5	32.9
reg	mrc:2	100	43.7	50.6	24.6	49.3	7.0	49.8
reg	mrc:2	200	42.8	51.2	24.2	50.5	7.0	50.3
reg	mrc:2	300	35.1	50.9	18.3	49.2	6.2	51.3
reg	mrcysdp:2	100	46.0	51.3	41.4	50.1	30.6	44.9
reg	mrcysdp:2	200	45.0	51.9	41.7	50.9	33.8	49.3
reg	mrcysdp:2	300	41.0	51.5	40.4	51.2	39.5	51.1
reg	mrm	100	33.2	50.6	8.3	50.4	11.3	49.5
reg	mrm	200	34.2	50.2	12.7	50.2	4.7	50.2
reg	mrm	300	28.9	50.7	7.7	50.0	6.6	48.9
reg	ms	100	33.9	51.3	24.7	51.1	16.8	50.4
reg	ms	200	35.8	47.4	24.1	49.0	15.2	49.1
reg	ms	300	36.6	51.3	23.2	51.0	12.7	51.0

Table 8.2. *Error rates (ERR) for all conducted training sessions on the UDXOR data set for nets of different sizes and input sequences of varying base length, excluding terminator symbol and sizes 2 and 4.*

The unstable behaviour for some rounds is a consequence of using no limit for the weights magnitude, thereby creating numerical instabilities. These "run-offs" could easily be handled which is however explicitly not done for comparing the different modes and nets: it can be seen that LSTM tends to run off more often then the Elman net. The Elman net runs off only in very few cases after ≈ 800 iterations of overfitting.

Regardless of numerical instabilities the assumption could be proven true that the periodic mode must have an advantage over the other modes. This is presumably due to having a translation invariant pattern set.

An interesting but unclear result is that the Elman net performs well in recursive mode, but failed in the periodic mode, which is the same as computing the given sequences in recursive mode but starting from different positions within the sequence. This might be explained by the absence of gates (as with LSTM) that prevent the weight updates from averaging out to zero.

8.3 The "ARCENE" data set

The ACRENE data set has been created from several sets of mass-spectrometric data of cancer and normal patterns and has been used at the NIPS[3] Feature Selection Challenge in the year 2003. The challenge was part of a workshop and its results are still available on the internet[4] where this data set can also be downloaded. The data had been manipulated for that challenge by removing certain elements, inserting others, removing the base line (which is roughly the pointwise minimum of all given patterns) and pointwise normalisation. The details can be found at the mentioned website[5].

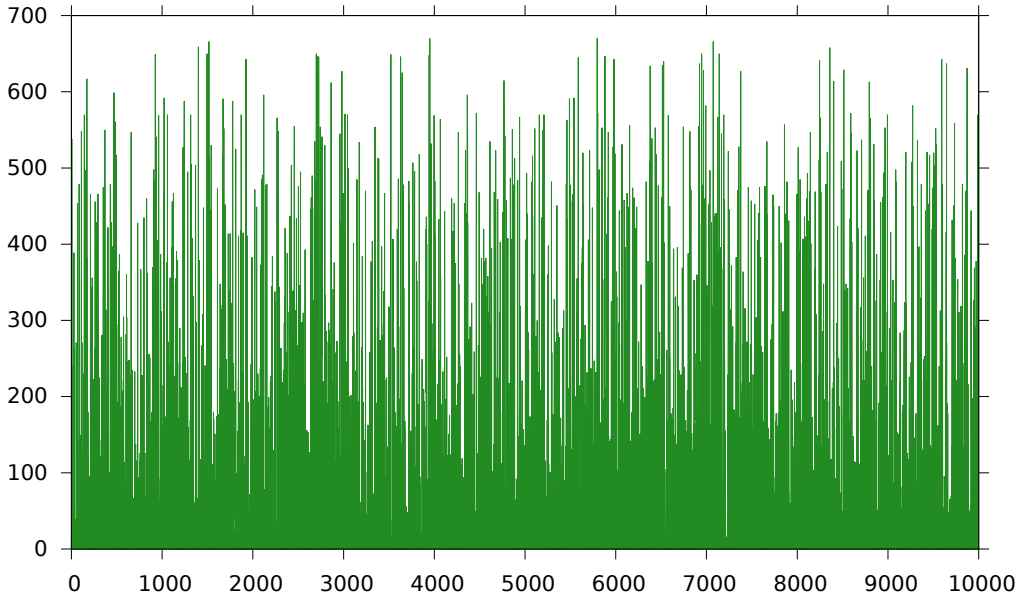

Figure 8.6. *Plot of one pattern from the ARCENE data set: x-axis shows the index of the input vector (with length $l = 1$), y-axis shows its value that is not normalised into the interval $[0, 1]$ yet.*

[3]Neural Information Processing Systems Foundation
[4]`www.nipsfsc.ecs.soton.ac.uk`
[5]`www.clopinet.com/isabelle/Projects/NIPS2003/Slides/NIPS2003-Datasets.pdf`

Each pattern is a sequence of fixed length 10 000 of scalar values from the interval $[0, 1]$. The plot of one example pattern can be found in figure 8.6, an unmodified example can be found in later sections in figure 8.13. The task is to learn to distinguish between cancer and normal patterns by mapping cancer patterns to 1 and normal patterns to 0.

Settings

The pattern set consists of 200 patterns and is divided into the training and testing set of 100 patterns each, consisting of 44 positive and 56 negative patterns each. The following setups are used in this chronological order:

i) At first, LSTM and an Elman nets have been trained, the Elman net uses $f(x) = 2\,\mathrm{sgd}(x) - 1$ as activation function for the hidden neurons. The size of each net varied between 1 and 3. No terminator symbol has been used for these setups. The initial weight span for each setups was 0.1. The iteration limit was set to 1000. Each setup has been repeated over 10 rounds (10 times). Cross entropy has been used as error function. For LSTM, the input gates have been biased to -3 for the first and stacking -0.1 for each additional block. RPROP$^-$ has been set to start with a learning rate of 0.1 for the gradient initialisation, factors 0.5 for dropping and 1.2 for increasing the steps, 0 for minimum absolute increment and 1.0 for maximum absolute increment. This setup has been combined with the sequential, recursive, periodic and D&C mode.

   All of these experiments failed by not approaching a small error rate or showing any convergence.

ii) Afterwards, a setup-set has been created differing from the first in the following properties: the stop criterion is met when the average MSE is below 0.1 and the maximum MSE is below 0.2. For the net size 6, the initial weight spans 0.1, 0.3, 1.0, 3.0, 10.0 have been defined and only the Divide and Conquer mode was used.

   This setup-set was used to figure out if only the weight span was to small or if the ARCENE data set was to hard to learn without additional preprocessing or completely different methods.

   Since the D&C mode started working for an initial weight span of 1.0, also the network sizes 5 and 4 have been tested on this weight span.

iii) After clarifying that a rather big initial weight span would be needed, the sequential mode, that is, the recurrent default, and the recursive restructuring mode have been used for training on this pattern set, each with an initial

weight span of 1.0 and a net size of 5 neurons. Settings for RPROP$^-$, biasing for LSTM and the stop criterion are used from setup ii). E_{SSE} has been used as error function. The sequential mode has be re-run with Elman and LSTM nets of size 9 to account for the larger amount of free parameters (weights) within the recursive modes due to the fanout increasing from $k = 1$ to $k = 2$.

iv) Finally the Divide and Conquer mode and its bidirectional enhanced version was tested. Since the previous setups did not let LSTM run successfully, some modifications were made: the stop criterion was met when the average MSE was below 0.05, the input gate biases were -2, -2.5 etc., the output gates were biased at -1, -1.33 etc. and the initial learn rate was set to 1. Also, the minimum step size for RPROP$^-$ was set to 10^{-6} and the cross entropy was used again as error function.

8.3.1 Results

Setups i) and ii)

As already mentioned in the description of setup i), all of those experiments failed to significantly converge towards a solution. Only the D&C mode tended to converge and only for the net size 3. The first interesting plots can be seen in figure 8.7 where the impact of the initial weight span on the D&C mode with nets of size 6 from setup ii) is shown: for the initial span of 0.1, a few plots tend towards 0 but a few stay on rather high MSE and ERR; the average ERR at the end is 11%.

For a weight span of 0.3 the plot-bunch looks a bit more compressed and the stop criterion is met earlier than before while the maximum MSE is lower at the end. The average ERR drops to 4%.

For a weight span of 1.0 the plot-bunch looks very compressed and all but one plot reach a low MSE while most of the rounds meet the stop criterion. A few rounds describe a common path in the plot. The average ERR stays at 4%.

With an initial weight span of 3.0 the trend changes. At the beginning of the training, peaks can be seen in the MSE. Only one round meets the stop criterion and each round seems to describe its own curve that has a somewhat constant pointwise distance to every other round. MSE and ERR still tend to drop towards the end where the average ERR is now 10%.

Finally, with a weight span of 10.0 the training is broken. After strong fluctuations and a peak MSE all plots join a path of almost constant behaviour. There is a small

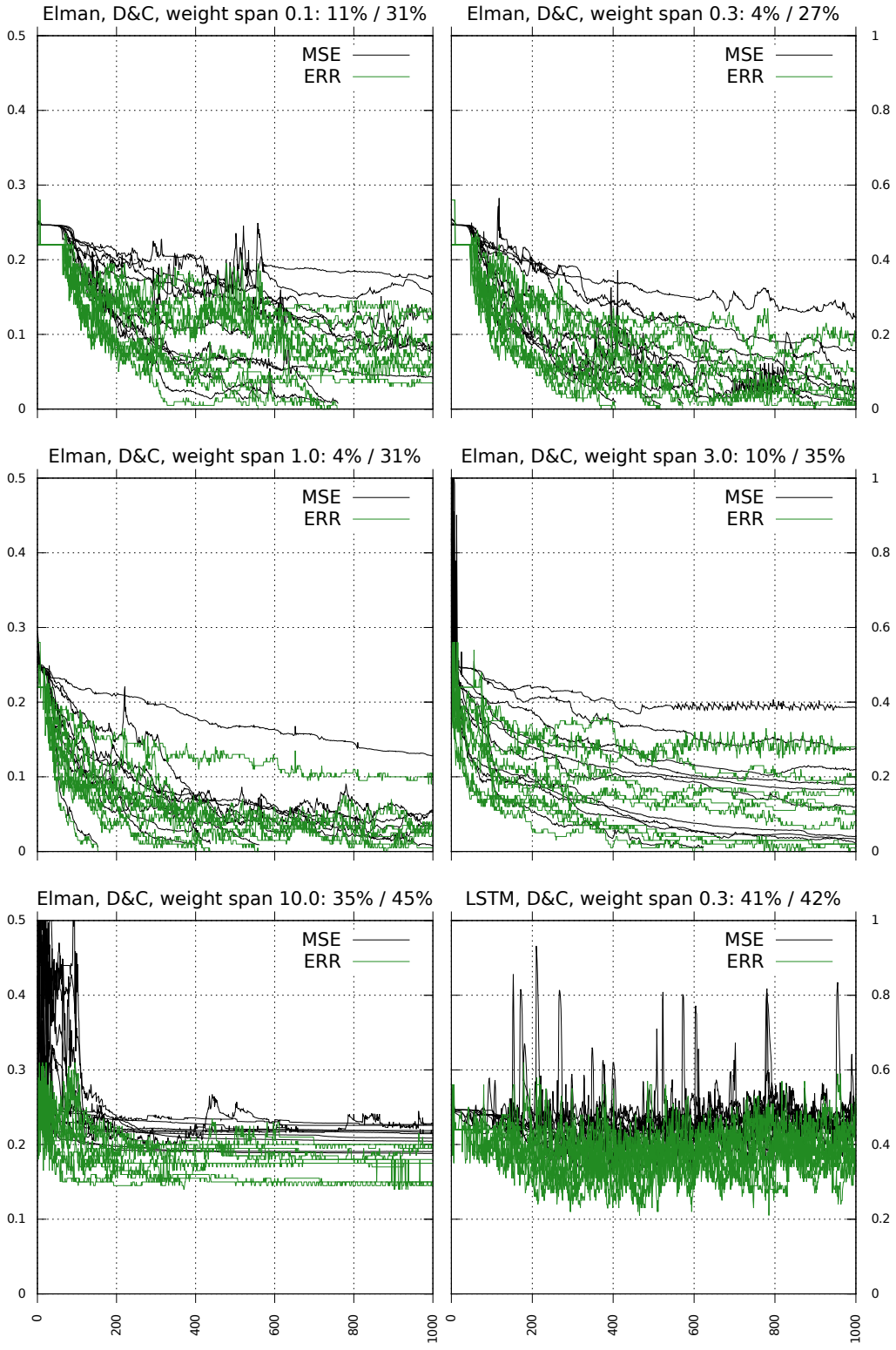

Figure 8.7. *Training graphs for D&C mode for several weight initialisation spans on the ARCENE data set. Only one example for LSTM is shown as they all look equal. The nets size for these experiments is* 6.

spread in the ERR that however does not change over time. The average ERR with 35% is close to guessing (remark that there are only 44% positive patterns).

One plot for the training session using LSTM is depicted. This however only shows that no convergence can be made out but strong fluctuations and peaks in the MSE occur.

Setup iii) and iv)

Training plots of the sequential mode (the recurrent default) and recursive mode with Elman nets and LSTM are found in figure 8.8. It also contains plots for the (bidirectional) D&C modes but without plots for LSTM as they are comparable to the one from sequential LSTM.

On the sequential mode the Elman nets quickly achieve a slight drop of the ERR within 100 iterations and then no significant changes occur so that the training is concluded with an average ERR of 26%. The lowest individual ERR is achieved by round no. 2 with 16%. Several rounds share the lowest ERR on the testing set with 29%. LSTM does not converge at all but produces wild peaks in the MSE with strong fluctuations in the ERR.

Using the recursive mode results in round no. 8 meeting the stop criterion and interpolation (0% ERR) but with a testing ERR of 41%. Many others still drop towards a low MSE and ERR. The most rounds show a tendency towards convergence within 200 iterations while a few remain almost constant for > 400 iterations. Only 2 rounds seem to not learn anything according to the MSE. Round no. 10 stops with 6% training ERR and achieves 26% ERR on the testing set. The recursive mode using LSTM does again not succeed but the plots are specific: the MSE tightly stays on a level of 0.25 for each round with short and few ticks above or below this level.

The Divide and Conquer modes achieve results comparable to those from the recursive mode. The plots for simple D&C show a rather even training progress. The training and testing ERR are slightly smaller with 14% and 29%. Bidirectional D&C shows a few more fluctuations and peaks and the resulting average ERRs are as those from recursive mode. The sequential mode with network size 9 does not show progress and even produces higher ERRs values than the sequential mode with nets of size 5. All ERR values are to be found in table 8.4.

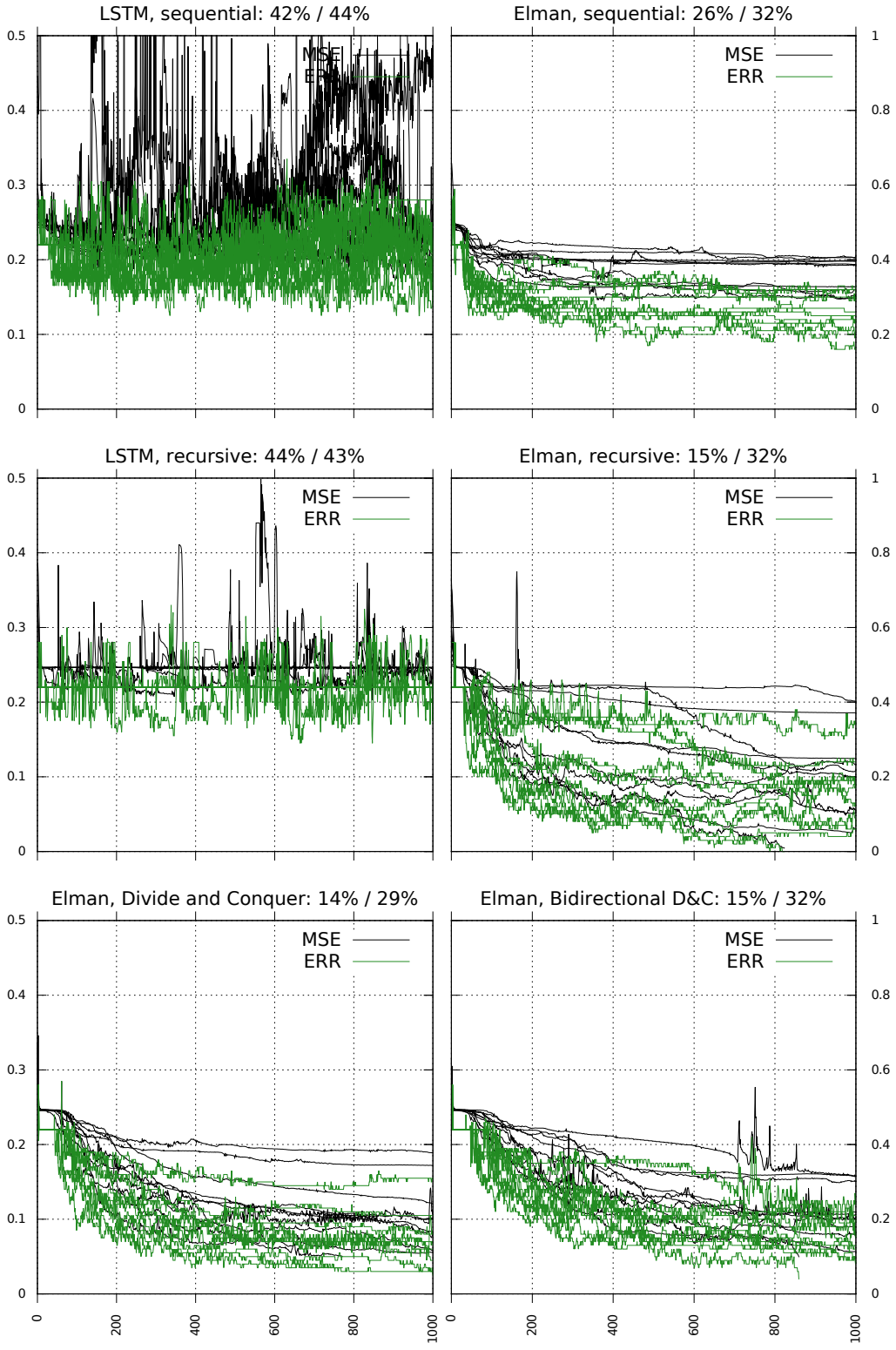

Figure 8.8. *Training graphs for sequential and the remaining restructuring modes on the ARCENE data set. The upper four plots result from setup iii), the two plots below belong to setup iv).*

8.3.2 Interpretation

At first the ARCENE data set seemed to constitute a classification problem that is to hard to learn by simply putting its data into a recursive net. The Divide and Conquer restructuring method was the best performing and robust method in the experiments conducted on the other data sets. It revealed that a bigger initial weight span was needed to approach the ARCENE problem.

LSTM did not succeed in any of this experiments, which is surprising. As a conclusion, LSTM, with the training settings like RPROP$^-$ used here, might not be well-suited for operating on real-valued data. Plain gradient descent with very small learn rate could be tested, however these experiments are not to compare different kinds of networks, but to compare the restructuring methods. It is therefore enough to have them working with the Elman nets. All training and testing ERRs for finding an appropriate initial weight span can be found in table 8.3.

As announced in chapter 3.3, the statement in [KP90] (Backpropagation is sensitive to initial conditions) can be reconfirmed in the sense that it does make no difference if the gradient computed by BPTT/BPTS is computed by RTRL instead.

net	mode	weight span	train.	gen.
reg	mrm	0.1	11.0	30.8
reg	mrm	0.3	4.1	26.8
reg	mrm	1.0	4.4	30.5
reg	mrm	3.0	9.8	34.7
reg	mrm	10.0	34.7	44.7
lstm	mrm	0.1	37.5	39.6
lstm	mrm	0.3	40.5	41.5
lstm	mrm	1.0	41.6	43.2
lstm	mrm	3.0	38.3	41.7
lstm	mrm	10.0	42.7	42.2

Table 8.3. *Training error (ERR) and generalisation error for Divide and Conquer method with a net of size* 6 *and varying initial weight span on the ARCENE dataset.*

The training ERRs (table 8.4) show that the restructuring modes, that is, both D&C modes and recursive restructuring, outperform the recurrent default using a recurrent Elman net. Increasing the size of the recurrent Elman net's hidden layer to 9 results in the net to have approximately the same amount of weights that could be adapted. This actually worsened the results for the sequential mode.

As a conclusion, the restructuring modes are not better just because they imply the used recursive net to have more weights (through a second layer in the weight tensor) that can be adapted.

All generalisation error rates are bad and cannot be interpreted in the way that the nets actually have satisfactorily learned the problem. This is expected as the ARCENE data set was specifically designed to be hard to approach.

However, restructuring the ARCENE data set from a set of sequences into a set of trees and processing these with recursive nets raises the confidence that it can be approached without heavy content-related preprocessing.

net	size	mode	train.	gen.
reg	5	mrm	13.6	29.3
reg	5	ms	26.0	31.5
reg	5	mrmbd	14.6	31.7
reg	5	mrc:2	14.5	31.8
reg	9	ms	29.0	35.5
lstm	5	mrmbd	41.6	41.0
lstm	9	ms	43.0	42.9
lstm	5	mrc:2	43.9	43.0
lstm	5	ms	42.0	44.3
lstm	5	mrm	39.6	44.6

Table 8.4. *Training and generalisation error (ERR) for sequential mode, recursive and Bidirectional Divide and Conquer with fixed weight span* 1.0 *on the ARCENE data set, ordered by net kind and generalisation error. Periodic mode is not included due to completely constant behaviour. Sequential mode with net size 9 is included to account for the implicit higher amount of free parameters (recursive weights).*

8.4 The "SCOP" data set

This Experiment bases on a part of the data set that was used in [HHO07] which dealt with data from the SCOP (Structural Classification Of Proteins) database[6]. Therein LSTM was used to classify genome sequences as belonging into a certain SCOP group. The SCOP group was either a "fold" or a "superfamily". The fold of a genome roughly determines the physical shape of the biological entity, most of those entities being proteins. The fold can be a helix, double helix etc. and below this level the superfamily is arranged. The family is the lowest level. This hierachy of nested groups allows to systematically arrange the huge amount of different genomes

[6] `http://scop.mrc-lmb.cam.ac.uk/scop/index.html`

found in nature (humans, animals, plants, ...) basing on expert knowledge; this expert knowledge is the physical shape of the biological entity (protein) that is encoded by the particular genetic sequence.

sequence	**class**
CKGKGAPCRKTMYDCCSGSCGRRGKC	1
PTVEYLNYEVVDDNGWDMYDDDVFGE- ASDMDLDDEDYGSLEVNEGEYILEAA- EAQGYDWPFSCRAGACANCAAIVLEG- DIDMDMQQILSDEEVEDKNVRLTCIG- SPDADEVKIVYNAKHLDYLQNRVI	0

Figure 8.9. *Example genome sequences from the SCOP data set. The letters encode amino acids. The first sequence is named "d1cnna_g.3.6.1 (A:) Conotoxin {Sea snail (Conus magus), M VIIc}" and has been selected from the positive training set, the second is named "d1doi___ d.15.4.1 (-) 2Fe-2S ferredoxin {Archaeon Haloarcula marismortui}" and selected from the negative testing set (raw data taken from the corresponding .fasta-files).*

As described in [HHO07], the data set is constructed by joining all except one family (i.e. all sets of genomes) that belong into the certain superfamily to define the positive training set. The positive testing set consists of the family excluded from the positive training set. The negative set is made out of pattern not belonging to the same fold as the (super)family. According to [HHO07][3.1.3: Training set] the positive training set size was increased using the PSI-BLAST algorithm to increase the amount of positive pattern.

The web link[7] to the downloadable pattern set can also be found on the web page of the Institute of Bioinformatics, Johannes Kepler University Linz.

The pattern set provided in the above link contains 102 SCOP superfamilies in total and they consume approximately 250 MB of memory in the plain text encoding that can be seen for example in figure 8.9. This leads to a very huge amount of necessary computing time.

To reduce the amount of data, only the superfamily g.3.6 has been considered from which the family **g.3.6.2** is withhold as the positive testing set. This pattern set takes up roughly 1 MB of memory for almost 5000 patterns in total (for details see below).

Because the other superfamilies are not learned on, this experiment cannot be compared with the experiments in the mentioned paper. Additionally the net size

[7] `http://www.bioinf.jku.at/software/LSTM_protein/`

in the following experiments is only 3 (i.e. 3 hidden neurons) and only Elman nets are used; the LSTM net used in [HHO07] consists of 13 blocks and an input layer of size $20 \cdot 11$ (therein referenced as "profile length 11") which receives the amino acid encoding of 11 consecutive elements of the respective genome. It was however not intended to make the following experiments comparable against those from the literature, but to compare the different restructuring methods against each other and the recurrent default.

Settings

The pattern set consists of 4765 patterns and is divided into the training set, consisting of 100 positive and 3313 negative pattern, and testing set, consisting of 28 positive and 1324 negative patterns each. Three setups have been created as follows that differ only in using the so-called balance gradient (see below) or not, the amount of iterations and the initial learn-rate for RPROP$^-$. All use the standard sigmoidal function sgd(x) as activation function.

i) Elman nets of fixed size 3 have been trained with an initial weight span of 1.0 (i.e. rather big). The iteration limit was set to 500. The setup has been repeated in 10 rounds and cross entropy has been used as error function. RPROP$^-$ has been set to start with a learning rate of 0.1 for the gradient initialisation, factors 0.5 for dropping and 1.2 for increasing the steps, 10^{-6} for minimum absolute increment and 1.0 for maximum absolute increment. The training has been stopped once the average MSE for two consecutive iterations dropped below 0.01 and the maximum MSE dropped below 0.1. This setup has been combined with the sequential, recursive, D&C mode and the periodic mode with terminator symbol.

ii) To account for the imbalanced amount of positive and negative pattern, training has been repeated using the balanced gradient (see below). This has be done basing on setup i) and starting with the respective final weight matrices of each round of the previous setup. Only 100 iterations have been used and the initial learn rate for the gradient initialisation of RPROP$^-$ has been set to 1.0.

iii) As balanced gradient could have been used from the beginning, setup i) has been repeated by doing so, and bidirectional D&C with and without terminator symbol has been included for completeness.

BER and Balanced Gradient

The amount of positive pattern share only 3% of the size of the total pattern set. Thereby a net which produces a constant output ≈ 0 would result in a classification

error of only 3%. As can be seen in the resulting plots in the next section, using the default gradient all nets ERR quickly reach a level close to $0.03 \approx 10^{-1.5}$ on which they stay for 50 to 200 iterations. Afterwards, the learning towards a non-constant solution starts. On a pattern set with 50% of positive and negative patterns, this first level of constant output is displayed as a horizontal line in the middle of the plot (MSE of 0.25 and ERR of 0.5).

Because the error rate (ERR) is not expressive for an imbalanced amount of positive and negative pattern, the balanced error rate (BER) can be used. For a pattern set M let $|M| = P + N$ where $P = |\{(x, y) \in M : y = 1\}|$ is the number of positive pattern and $N = |\{(x, y) \in M : y = 0\}|$ the number of negative pattern. Further let ERR_P be the error rate on the positive set (i.e. the fraction of false negative) and ERR_N be the error rate on the negative set, that is, the fraction of false positive. The **BER** is defined by

$$\mathrm{BER} = \frac{1}{2}(\mathrm{ERR}_P + \mathrm{ERR}_N)$$

as the mean of both indicators. As a conclusion, a low BER can only be achieved if the relative amount of false positives and false negatives is balanced and small. If by $P = N$ there is no imbalance, it is ERR = BER.

For computing the BER, the patterns are just grouped by class and the respective ERRs are scaled and summed up. The ERR of the false positives, i.e the negative patterns falsely classified as positive, is scaled by the factor $(N + P)/2N$. Because of $P < N$ in this experiment, this is a scaling downwards. The ERR of the positive patterns (that are fewer in amount) that have falsely been classified as negative is scaled up by the factor $(N + P)/2P$.

This scaling can also be interpreted as to happen within the error function E and from there it can be interpreted to simulate a pattern set that contains rational multiples of the patterns with the scaling factors as mentioned before. During training this would have the same impact to the gradient as to the error rates. As a conclusion, the **balanced gradient** is defined by

$$\frac{\partial E}{\partial w}^{(\mathrm{bal.})} = \frac{1}{2}\left(\frac{\partial E}{\partial w}^{(\mathrm{pos.})} + \frac{\partial E}{\partial w}^{(\mathrm{neg.})}\right)$$

as the mean of the gradient on the positive and negative pattern set. Remark that according to section 3.4 each gradient is normalised according to the amount of pattern that are part of it. Therefore by creating subgroups and applying the above formula a different gradient is created.

As gradient descent is applied on a non-linear function, it cannot be expected that the results from using balanced gradient from the beginning on are the same or of the same quality as those acquired by retraining after using the standard gradient. The following results will show how the different approaches compare to each other. Using balanced gradient can be understood as changing the error measure.

The plots still contain the unmodified MSE and ERR values while the (average) BER is specified within the heading of each plot and the concluding table at the end of this section.

8.4.1 Result

The reasons that lead to using the BER motivates to also accommodate the plots of the training sessions. Therefore a logarithmic scale is used in the following plots that however still show the ERR values. As a side-effect, the ERR-plot for each round can be unambiguously assigned to the MSE-plot of the same round as the magnitude of the ERR and MSE values differ more strongly between different rounds then between each other.

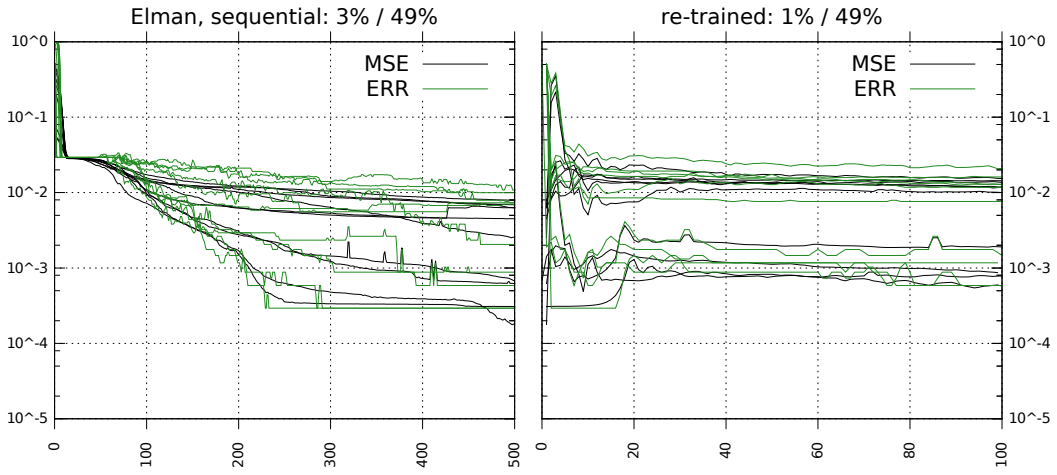

Figure 8.10. *Training graphs for sequential mode for the SCOP data set using setup i) (left) with subsequent retraining towards the balanced error rate through balanced gradient using setup ii) (right).*

Setups i) and ii)

In standard sequential mode (figure 8.10) the training succeeded with 3% average BER after 500 iterations but resulted in no generalisation (49% BER). Re-training

with balanced gradient reduced the BER to 1% after 100 iterations but did not change the generalisation quality. The stop criterion is not met. According to the plots, the rounds consist of one group that remains on a rather high ERR of 10^{-2} and some rounds that approach an ERR of 10^{-3}. Changing the error measure (compare left plot against right plot) strongly increased the error rates during the first 10 to 20 iterations. In both cases, no round reached a state of interpolation.

After recursive restructuring (figure 8.11) the BER on the training set was 7% and on the testing set it was 42%. The stop criterion was met in some cases. Re-training reduced both BERs to 2% and 35%. During re-training, each round stays within a certain order of magnitude and produces only small changes within this magnitude (MSE and ERR). The first training seems to have produced a spread amongst the different rounds. The training BERs are bigger than those from sequential mode, but the testing BERs are smaller. A few rounds have reached a state of interpolation with a very small MSE of approximately 10^{-4} and an ERR of 0%.

Also for recursive and periodic restructuring the re-training with balanced gradient strongly increased the error rates for a few iterations, after which they quickly return to lower values.

The periodic mode did not produce a spread amongst the different rounds. The training BER increased from 1% to 5% after re-training while the testing BER dropped from 40% to 37%. No interpolation was reached.

The Divide and Conquer method resulted in 0% training BER after training and re-training. During the first training, the stop criterion was met for each round within less then 250 iterations and changing the error measure through balanced gradient increased the error rates only a bit within the first iterations of re-training. During re-training, the stop criterion was also met in all but one rounds. In both cases the error decreased very quickly. The testing BERs are 48% and 47%.

mode	**train.**	**gen.**	**train.**	**gen.**
	initial		**re-trained**	
mrc2	6.9	41.9	2.4	35.5
mrcysdp2T	0.7	40.4	5.3	37.1
mrm	0.0	47.6	0.0	46.8
ms	3.3	49.2	1.1	48.9

Table 8.5. *BERs after training with setup i) and ii) for the SCOP data set.*

Setup iii)

When training with balanced gradient from the beginning on (figure 8.12), the sequential mode behaves almost similar and reaches training and testing BERs

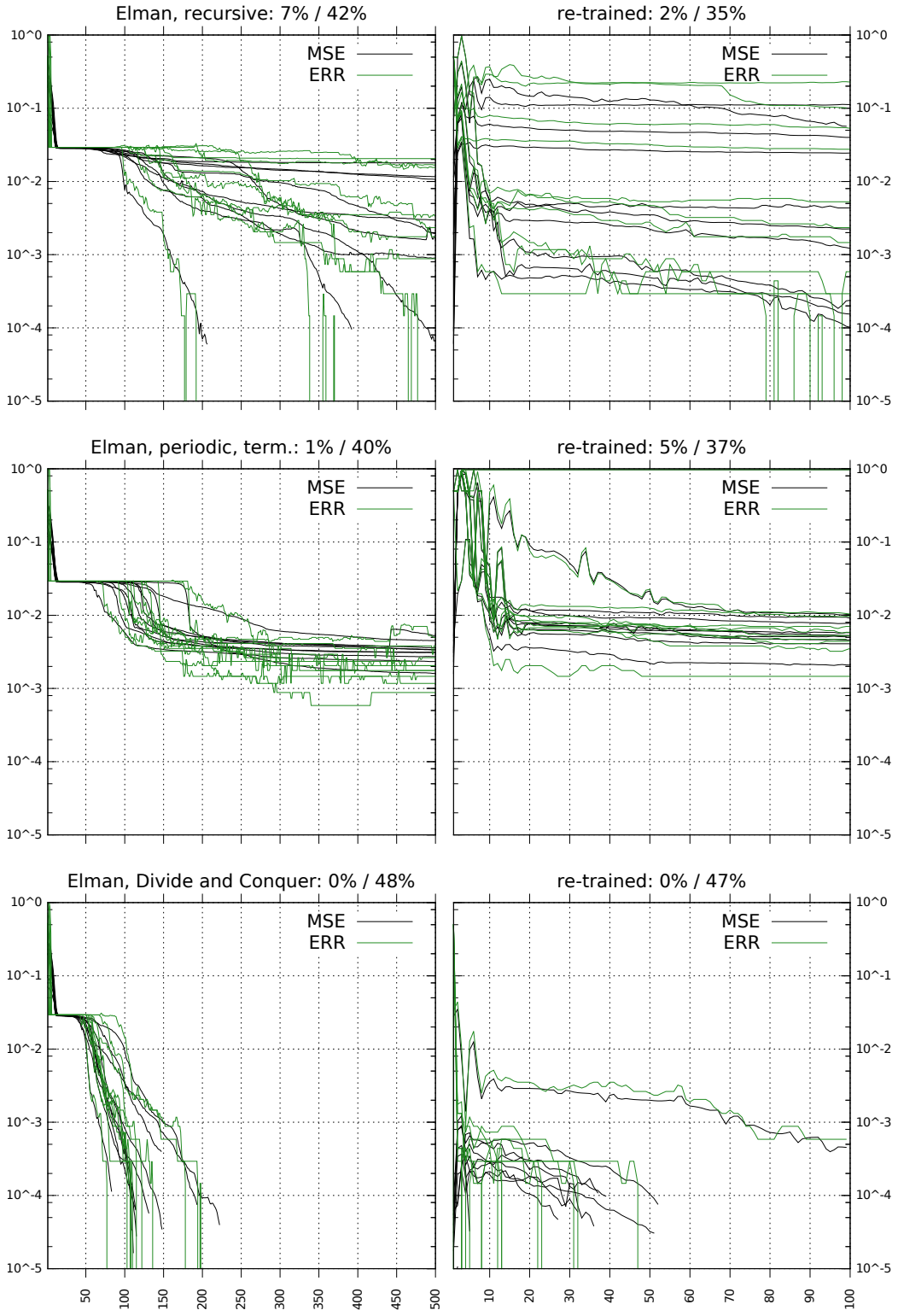

Figure 8.11. *Training graphs for restructuring modes for the SCOP data set with setup i) and ii).*

that are between those from the previous setups, that is 2% and 48%. Significant differences are also not found during the training processes of the restructuring modes, however the resulting BERs differ.

Using the recursive mode with balanced gradient results in a slightly faster drop of the error rates when comparing both plots (remark that only 300 iterations are used in this setup while the previous were using 500 and 100).

A significant drop in the BER can be found on the periodic mode (here only tested together with terminator symbol in all setups): the BER on the test set is only 31% with a training BER of 1%. The average ERR on the positive testing set is $\approx 63\%$ after retraining and $\approx 60\%$ after fully training with balanced gradient. Through the few amounts of positive pattern, this strongly decreases the testing BER. The round nr. 7 in periodic mode in setup iii) reached a testing BER of only 7%. It was the only round whose results strongly deviated from the others. Its MSE is inconspicuous with $\approx 0.0418 \approx 10^{-1.38}$ and does not reveal this low BER; in fact, it has the highest ERR of $\approx 5\%$ on the negative training and testing sets. For comparison: after retraining, the best BER achieved by a single round is found in round 1 in recursive mode with a BER of 32%.

One remarkable fact is that the difference between the error measures (BER and ERR) can be seen in the first 100 iterations of the periodic mode. The plot displays the ERR, that is, the non-scaled fraction of erroneous classified pattern. This fluctuates quickly within two orders of magnitude, the BER however would fluctuate only slightly. During this state, the net changes between a constant output greater or smaller then 0.5, resulting in either 97% correct or 3% correct patterns because of the imbalanced amount of positive patterns. The BER is 50% in both cases.

Bidirectional Divide and Conquer with and without terminator symbol have been tested only with this last setup. Both methods show similar plots like those from Divide and Conquer. All D&C-like methods achieve 0% training BER, but judging on the testing BER, the bidirectional one outperforms the plain D&C, dropping BER from 47% to 45%, and additionally using a terminator symbol outperforms the mode without it, dropping the BER to 42%, compare table 8.6.

8.4.2 Interpretation

Table 8.5 shows that the sequential mode (“ms”) is outperformed by all other modes. The best performing mode after re-training according to setup ii) is recursive (“mrc2”). It reaches the lowest testing BER of 35.5% with a training BER of 2.4% which is higher then the training BER of 1.1% of the sequential mode. This could

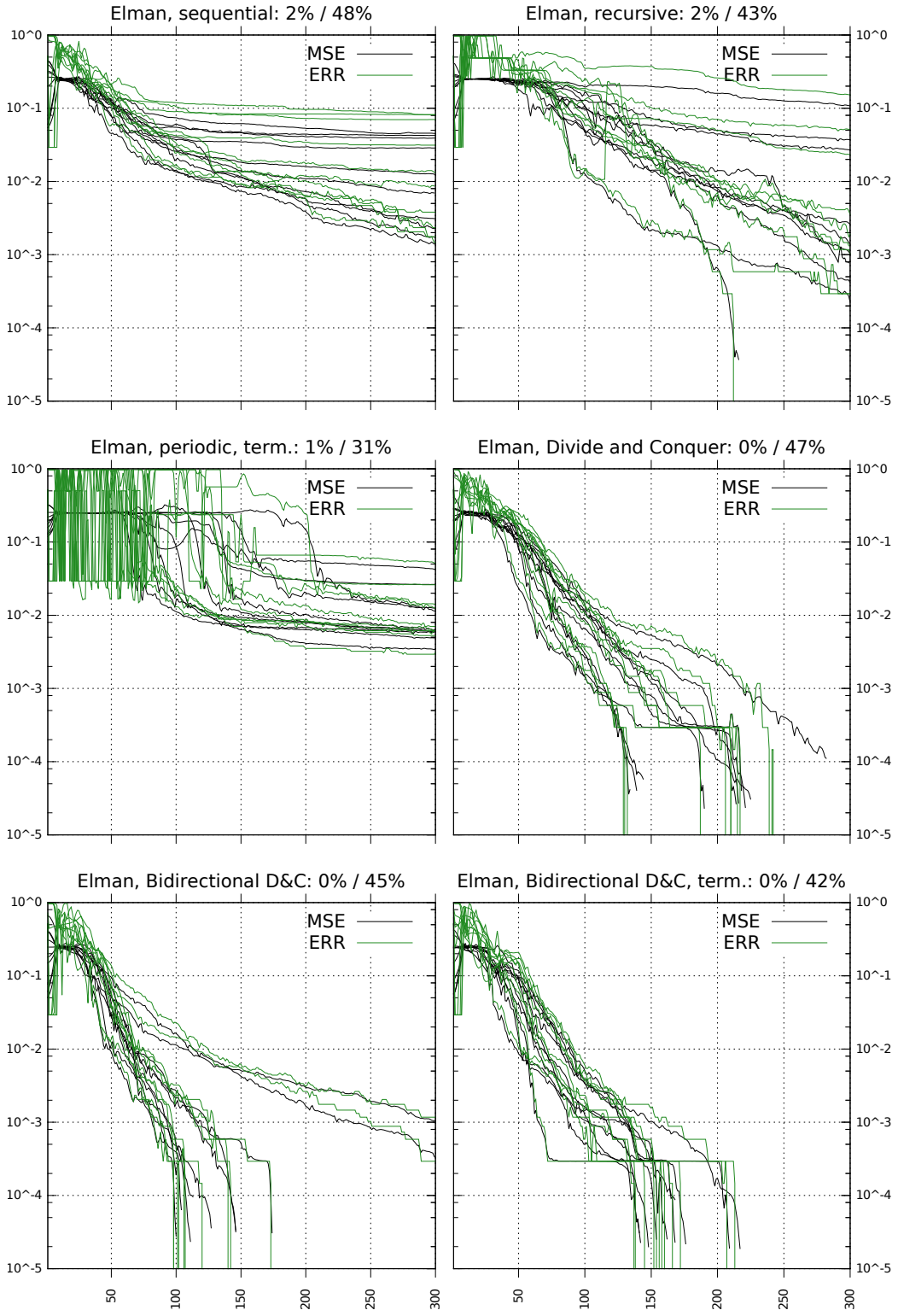

Figure 8.12. *Training graphs for sequential and restructuring modes on the SCOP data set using the balanced gradient of setup iii).*

be interpreted in the way that the sequential mode was overfitting for to long. However, the D&C mode ("mrm") reaches lower testing *and* training BER.

According to table 8.6 the previous considerations must not even be made because of the last setup. No mode results in a training error that is higher then the one from sequential mode. All restructuring modes result in a lower balanced error rate, which is however still very high. Using a net with more then 3 hidden neurons and a recursive MLP would probably result in better numbers.

One single round in the periodic mode achieved a competitive small generalisation BER of 7%. When adjusting the error rates to ignore the outlier, the periodic mode in the last setup outperforms all other modes with 34% BER. The difference to the recursive mode from re-training, 35.5%, is however not big enough to conclude that an intrinsic property of the data set has been found. This could, for example, be expected when dealing with cyclic genes.

Through the balanced gradient, the ERR increased shortly after starting the re-training in setup ii) as could be expected. Previous small magnitudes of MSE and ERR values where reached quickly, which could be expected because of the gradient initialisation described in section 3.4. Because the initial weight matrices are not randomised but taken from the previous setup, the first gradient contains values of very small magnitude for patterns of the negative class. Their gradients ratio gets decreased because it was small within the last gradient of the first setup. Only the positive patterns' gradient is scaled up, but also their magnitude is small. Changing the error measure (the error function) by weighting the different patterns does not completely ruin the previously learned weights, they just need to be readjusted. This can be understood by the following example:

Assuming that one weight has been trained to an exemplary value of $w_i = 0.05$, the default RPROP$^-$ method would take the sign of the gradient which is assumed to be $\partial E/\partial w_i = -10^{-4}$ and compute the update $\Delta w_i = -1.0$, because the initial learn rate is set to 1.0. It thereby disturbs the previous weights value to $w_i = -0.95$.

mode	**train.**	**gen.**
mrcysdp2T	0.7	31.3
mrmbdT	0.0	41.9
mrc2	1.8	43.4
mrmbd	0.0	45.4
mrm	0.0	47.0
ms	1.8	48.1

Table 8.6. *BERs after conducting the training session according to setup iii) on the SCOP data set.*

An initial learn rate of 0.01 would result in the update -0.01 and still result in a perturbation to $w_i = 0.04$. But here, the gradients magnitude determines the first step size to 10^{-4} resulting in the update -0.0001 and $w_i = 0.0499 \approx 0.05$. This happens even though the upper limit for a weight update is set to 1.0 so that later updates are not restricted in size by the magnitude of the gradient.

As a conclusion, the balanced gradient is a valid method for dealing with unbalanced pattern sets. It equals the default gradient for balanced pattern sets and the implicit scaling effect is proportional to the magnitude of imbalance. Therefore it could replace the default gradient completely.

8.5 The "NCIOvarian" data set

This data set consists of original, unmodified mass-spectrometric data that was used in the ARCENE data set. The patterns have been acquired directly from the website of the Center for Cancer Research, National Institute of Health, USA[8]. The "Ovarian Dataset 8-7-02" has been used. Its files have been split into a training set consisting of 81 positive and 46 negative ("control") patterns and a testing set consisting of 81 positive and 45 negative patterns. Each pattern is a sequence of length 15154 of vectors $\in \mathbb{R}^2$. Each vectors first component is the mass-to-charge ratio ("M/z") while the second value is the intensity at this particular M/z value. An example plot of such a pattern can be found in figure 8.13, it suggests that the M/z values are not increasing strictly linear but are slightly curved. All values are normalised into the interval $[0, 1]$.

Settings

Elman nets of fixed size 5 have been trained. The initial weight span was set to 1.0 and an iteration limit of 1000 was chosen. The setup has been repeated in 10 rounds and cross entropy has been used as error function. RPROP$^-$ has been set to start with a learning rate of 0.1 for the gradient initialisation, factors 0.5 for dropping and 1.2 for increasing the steps, 10^{-6} for minimum absolute increment and 0.1 for maximum absolute increment. The training has been stopped once the average MSE dropped below 0.05 and the maximum MSE was below 1. This setup has been combined with the sequential, recursive, D&C and bidirectional D&C mode each with and without terminator, and with the periodic mode only with terminator symbol.

[8] `http://home.ccr.cancer.gov/ncifdaproteomics/ppatterns.asp`

The stop criterion has been chosen with a limit for the maximum MSE that is practically no restriction as the MSE cannot be > 1. Therefore, only a somewhat small MSE of 0.05 in average must be reached for the training to stop and severe overfitting as in many of the previous experiments should be avoided as a consequence.

8.5.1 Results

The plots of the D&C methods can be seen in figure 8.14. All other modes resulted in constant output within each round, except for the recursive mode with terminator symbol: the rounds 4 and 10 of that mode (not depicted) did not behave completely constant. Round 4 achieved a training ERR of 12.6% and a testing ERR of 18.3%, however round 10 achieved 63.8% training ERR and 45.2% testing ERR.

The plain D&C training does not meet the stop criterion for 4 rounds, when attaching a terminator symbol, it is not met for 3 rounds. Switching to bidirectional D&C without terminator symbol reduces this to 2 rounds which do not meet the stop criterion. Finally, within the bidirectional D&C using a terminator symbol, all rounds stop through meeting the strop criterion. The training and generalisation error rates are consistent with these results which is summarised in table 8.7.

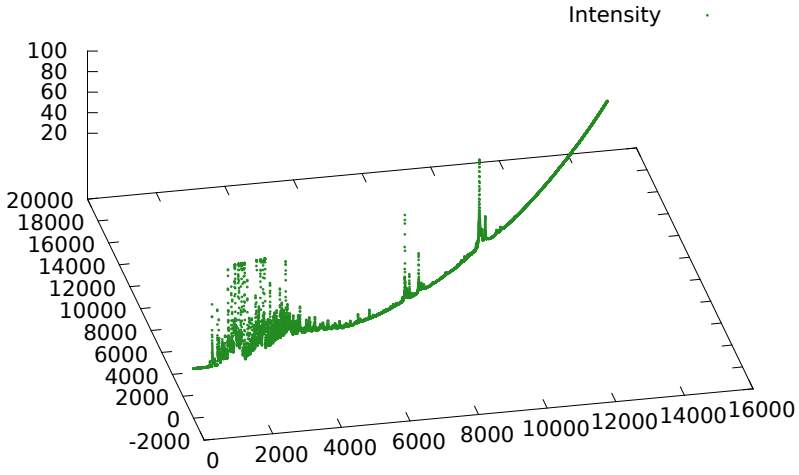

Figure 8.13. *3D-plot of one pattern from the NCIOvarian data set: lower axis shows the index of the input vector, left axis show each vectors first component and the upper axis the second. This is a positive (class* 1*) example of the training set.*

In general the training can be considered fast: for the best working mode, the iteration stops after 115 iterations for the fastest round.

net	mode	train.	gen.
reg	mrmbdT	5.1	8.6
reg	mrmbd	8.0	10.9
reg	mrmT	9.9	11.0
reg	mrm	12.8	15.5

Table 8.7. *Training and generalisation error for the NCIOvarian dataset.*

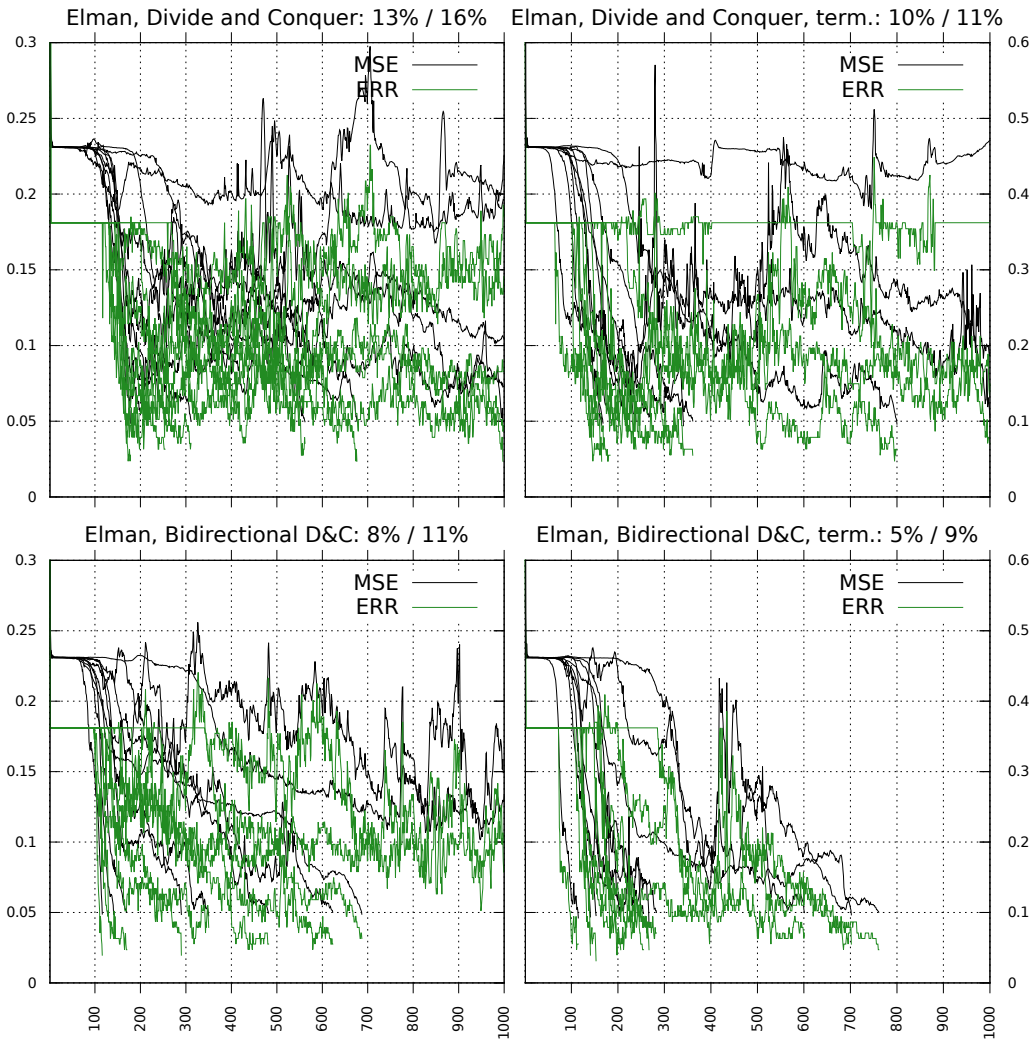

Figure 8.14. *Training graphs for (Bidirectional) Divide and Conquer, with and without terminator symbol on the NCIOvarian data set. Sequential, recursive and periodic mode each produced completely constant behaviour.*

8.5.2 Interpretation

Having practically all but the D&C modes fail is unexpected. It however shows the general robustness of this mode. The fact that enriching the sequences with a terminator symbol speeds up the convergence and increases the quality (in terms of error rates) confirms this kind of formal preprocessing as a valuable addition to the restructuring methods.

Achieving competitive low error rates on the testing sets proofs that overfitting is not a built-in problem to restructuring in general.

For this experiments, the default sigmoidal function has been used as activation function, while for the ARCENE data set, it has been modified. As the sequential mode fails, together with all but the D&C strategies, the conclusion can be made that for this kind of very long, real-valued data the function $2\,\mathrm{sgd}(x) - 1$ and presumably the hyperbolic tangent are better suited. Having the D&C modes being robust against changing the activation function is however a good and interesting result.

8.6 Overall summary

The conducted experiments can be considered an overall success. For each specific pattern set the default (sequential) processing using a recurrent net was outperformed by at least one recursive net that learned on the restructured input.

On the data sets containing symbolic data the periodic mode was the best performing while it failed for the data sets containing real-valued data. However, investigating individual rounds of the periodic mode revealed that, if the round is a success, it is of very high quality regarding the generalisation.

A pump-up effect for symbolic data (UDXOR) could be made out using the periodic mode. This can be interpreted as an indicator for an intrinsic property: by definition, the UDXOR data set is not translation invariant. This is due to the fixed positions where the relevant symbols are placed. Obviously their position is not relevant when considering the logical background and if the generating function of the pattern set were not known, one could now deduce the fact that the pattern set should be considered translation invariant as actually moving its elements in cycles does not prevent learning.

The Divide And Conquer method surprises with high robustness against changes in the kind of data set and an overall quick and straight learning. This might be

caused by the fact that for a given structure, each path from the root down to a leave constitutes a part of the input that can be understood as the view of a logarithmically distorting lens. Within the original sequence, the distance from the root node to one of its children is roughly 1/2 the length of the sequence, while the distance from from the child node to one of its children is roughly $(1/2)^2$ of the original length, and so on. This seems to equalise the contractive nature of a neural nets activation function while transferring the learning problem to finding the proper way of combining these logarithmic views (via two layers in the recursive weight tensor in opposite to one layer for a recurrent net).

Changing the activation function from $f_1(x) = \mathrm{sgd}(x)$ to $f_2(x) = 2\,\mathrm{sgd}(x) - 1$ on the ARCENE data set revealed the recursive mode being sensitive to the activation function. While using $f_2(x)$ almost every round produced a good result, using $f_1(x)$ produced almost constant behaviour with very few rounds starting to converge at the end of the training.

LSTM worked fine on symbolic data but not on continuous (real-valued) data. This might be a result of using other weight update schemes (RPROP$^-$) then in the cited papers (plain gradient descent). This could be a result of the LSTM-specific gradient calculation using so-called truncation but this question is not investigated any further.

The UDXOR data set has been learned with nets of size 3 and 5 and the ARCENE data set has been additionally learned with the recurrent default using a recurrent net of size 9 in comparison to the recursive nets using 5 hidden neurons. In all cases, the bigger recurrent net approximately has the same amount of weights (free parameters) as the smaller recursive net. However, the recurrent net was still outperformed. As a conclusion, learning on restructured input does not work better just because there are more free parameters to adapt.

Restructuring as formal preprocessing can help in dealing with the fading gradient. This is due to Elman nets being able to learn on data containing long term dependencies, probably because the iteration length (the path from a leave to the root) is strongly shortened from n to $\log(n)$.

Finally, the balanced gradient turned out to be an easy to use approach on imbalanced pattern sets and the weight initialisation for RPROP$^-$ proved useful for scientific purposes.

9 Conclusion

Restructuring sequential data can make a classification task more easy to handle by neural nets. It can be applied without analysing the content of the sequential data at hand. The experiments show (8.6) that it is worthwhile to examine machine learning on the restructured data. Even though the structure is arbitrary and not necessarily adapted to the problem at hand, better training results can be achieved on data that seems more complex. It is also shown that very small nets can lead to overfitting even on large input, that the activation function does matter and that a simple architecture like an Elman net sometimes succeeds where novel, more complex architectures like LSTM do not.

While the D&C mode was shown to be robust against the kind of input data (symbolic or continuous) and the activation function, the recursive mode turned out to be sensitive to the kind of activation function. LSTM surprisingly worked well only for the symbolic data, but not for the continuous data, where Elman nets were still trainable. As a conclusion, restructuring can help in dealing with long-term dependencies for classification tasks. Bidirectional restructuring, introduced through the asymmetrical convolution product between trees, was combined together with the D&C mode which resulted in even better results in many cases, while for one case plain D&C was working better.

The periodic restructuring mode was highly diverse, resulting in excellent results on symbolic data with LSTM, but not Elman nets (on the UDXOR data set), excellent results on symbolic data with Elman nets on the SCOP data set, but a complete failure on the pattern sets using continuous data.

By defining fixed frame conditions, the restructuring modes have proven to produce lower error rates than the recurrent net when learning for the exact same amount of iterations, as well as when stopping the training upon reaching a certain mean squared error. In the first case, the training error was generally outperformed, in the second case, this held also for the generalisation error. By additionally testing against a recurrent net of a size that results in approximately the same amount of free parameters as the recursive net, it was also shown that learning on the restructured input using a recursive net is not more easy just by having more free parameters to adapt.

Training on imbalanced pattern sets turned out to be quite easy when using the balanced gradient. The balanced gradient has been applied for re-training previously trained nets. This was achieved by relying on the local behaviour of RPROP$^-$ in combination with gradient initialisation. However, starting with balanced gradient from the beginning on resulted in slightly better results.

This thesis is not supposed to exhibit the restructuring methods as a replacement for anything. Neither is it supposed to be complete regarding the kinds of restructuring methods or the different principles, the abstract constraints, under which restructuring can take place. One principle would be the *empty shell idiom* that can be seen in recursive and periodic mode, where the structure does not contain any information inside, but only at its borders (the leaves). This principle can easily be violated for good reasons (see chapter 10).

Final remark

In this thesis the concept of restructuring sequential data has been proposed in order to apply machine learning methods on data for which they have not originally been designed. Motivated by bidirectional nets, still popular to use with neural nets, I intended to give an entry into a novel and promising field of study that can easily be extended. I did not intend to have shown certain architectures being superior or inferior to certain restructuring methods but rather that there are many reasons for why they can be combined into an even more powerful tool.

10 Future work

Many ideas for other restructuring methods have not been dealt with and popular tasks for machine learning like speech recognition or protein structure prediction have not been approached. As a conclusion, there is plenty of interesting and promising research that can be done using restructuring methods in the future. For example, methods for rule extraction could be utilised or transferred to recursive nets trained on restructured input (see [Jac05] for a survey). Examining the states of the net within subtrees of the restructured tree should give insight on how certain parts of the sequence influence the output, but without considering the sequence from the beginning on.

At first, a few experiments are mentioned that have not been reported because they have not been conducted extensive enough to make solid conclusion from them. This concerns the periodic mode with linear hidden activation function and the same mode with sigmoidal activation function and resembling a translation invariant classifier:

Linear activation function for periodic mode

A few experiments have been conducted using a linear activation function to examine how expressive the periodic mode then is. This was motivated by the comparison to the Fast Fourier Transformation. The unrestricted output of the hidden neurons result in an exponential growth of the activations because for a fan-out 2 the numerical magnitude approximately doubles for each layer. This required to start with very small weights. The few experiments however did not succeed. It would be interesting to find out if the linear periodic mode is more expressive when inserting non-zero inner labels.

Translation invariant classifier

The error function E^C for a translation invariant classifier as described in chapter 6.5 has been implemented and a few experiments have been run using this in combination with cross entropy. While for the UDXOR data set this mode seems to behave almost similar to the periodic mode, it behaved worse on experiments on

the ARCENE data set. This could be examined in more detail and cross entropy should be avoided at first.

10.1 Theoretical questions

While recurrent and recursive MLPs using the sigmoidal activation function are known to be approximation complete, meaning they can approximate arbitrarily well, it seems clear that any recurrent net operating on x can be approximated by a recursive net operating on ${}^{(r)}\widehat{x}$ or ${}^{(m)}\widehat{x}$. It however is unclear how the necessary sizes of the hidden layers correlate to each other. What kind of functions can be emulated without changing the hidden layers size?

Is seems obvious that a recurrent net can make computations on a sequence that cannot be emulated by a recursive net on the restructured sequence if they are of the same size. For example, a recurrent net can easily be defined that resembles the iterations of the well-known logistical equation, for example with $\mu = 4$ as fixed parameter, creating a mapping $f : [0,1] \to [0,1]$ with $f(x) = 4x(1-x)$. Iterating this equation by $x_{n+1} = f(x_n)$ can result in a chaotic behaviour of x_n. Starting this iteration with $x_0 = \sqrt{2} - 1$ results in a non-periodic iteration: there is no $n \in \mathbb{N}$ such that the n-th concatenation of f results in this (or any later) value: $f^{(n+m)}(x_0) \neq f^{(m)}(x_0) \forall n \in \mathbb{N}, m \in \mathbb{N}_0$. This can be seen by considering arguments of the form $a_n\sqrt{2} + b_n$ with $a_n, b_n \in \mathbb{Z}$. The series a_n generated by $f^{(n)}(a_1\sqrt{2}+b_1) = a_n\sqrt{2}+b_n$ is strictly monotonic increasing in absolute value. As a conclusion, the recurrent net emulating this iteration, for example by being activated on the input sequences $x_1 = (\sqrt{2}-1)$, $x_2 = (\sqrt{2}-1, 0)$, ..., $x_n = (\sqrt{2}-1, 0, \ldots, 0)$ etc. creates a unique output for each x_n. Through recursive restructuring, the recursive net would only have $log_2 n$ iterations to achieve this output by recursively processing sequences $(\sqrt{2}-1, 0, \ldots, 0)$, $(0, \ldots, 0)$ and $(\ldots, \perp)$ etc.

Vice versa it seems unlikely that a recurrent net can emulate a tree automaton (compare [CF01]) using a state space of the same size, even when the tree automaton receives input only at leaves. What kind of tree automata *can* be emulated?

10.2 Points of error injection

The pattern sets used in this thesis only allow to map structures to a single vector. This target vector however does not need to be assigned as output vector only in the structure's root. It could be worthwhile to examine whether using error

injection in every node of a restructured sequence is making the learning task more easy. This would create multiple gradients for the same classification task. A weighting scheme for the different gradients of the same pattern should be created. In general, the effective gradient per pattern should be normalised as such that the numerical magnitude per weight is the same as when injecting error only once. This is done in this thesis for the periodic mode. When a sequence x is "Divide and Conquer"-restructured into ${}^{(m)}\widehat{x}$, error injection in a leaf might result in the disability to have the error converge against zero upon training (because the label of the leaf does not contain enough information). As a conclusion, the weighting scheme could use the weighting factor 2^{-2d-1} for subtrees at depth d. This results in the sum of the weighting factors being close to 1 while all subtrees of the same depth d together have a summed weight of at most $2^d \cdot 2^{-2d-1} = 2^{-d-1}$. That means, the smaller the individual part of the input sequence is, the smaller the influence of the whole layer is to the collective weighting factor.

As described for the periodic mode, the target output for each tree does not need to be the same. Each tree can have its own output that can be a function of the whole sequence. This allows to learn sequence-to-sequence-mappings where the last element of the input sequence may influence the first output of the target sequence.

Furthermore, using the periodic mode, also for each level in the merged graph a target output can be assigned. That is, the target output may depend on the context visible for the current subtree. As a consequence, for a sequence of length 2^D, D output sequences of length 2^D can be assigned as target output, for example enabling the user to merge sets of overlapping sequences to save computing time when learning context free grammars as mentioned in chapter 6.5. All points of error injection, whether using a special weighting factor or not, can be interpreted as belonging into the specific error function $E(m)$ per pattern.

10.3 Sensitivity to the activation function

Using different activation functions for the reports in this thesis was rather arbitrary and based on experiences gathered from less thoroughly conducted experiments. It however revealed that some restructuring methods make the recursive net sensitive to the activation function when operating on the restructured results. More precisely: the recursive mode operated well on the ARCENE data set when using a centered sigmoidal function $(2\,\mathrm{sgd}(x) - 1)$ but failed most of the time when using the standard sigmoidal function. One could expect that the net adapts to these different functions by changing the bias. But it seems that having a state space in $[-1, 1] \subset \mathbb{R}$ is more

suited towards the recursive restructuring than having only $[0,1] \subset \mathbb{R}$. This should be examined in more detail.

10.4 More restructuring

Some potential, new restructuring methods have already been indicated in this thesis and some of the ones already proposed can easily be modified.

FFT restructuring method

From comparing the periodic mode with the Fast Fourier Transformation and its implementation through the Sande and Tuckey algorithm, it is obvious that for pattern with a length 2^n a "FFT" restructuring mode can be defined by changing the recursion formula

$$\begin{aligned} y_{r,k}^{(m-1)} &= y_{r,k}^{(m)} + y_{r,k+2^{m-1}}^{(m)}, \\ y_{r+2^{n-m},k}^{(m-1)} &= (y_{r,k}^{(m)} - y_{r,k+2^{m-1}}^{(m)})\epsilon_m^k \end{aligned}$$

by applying the arguments for $. + .$ and $(. - .)\epsilon_m^k$ to recursive nets. Either two recursive nets with fanout $k = 2$ or simply one recursive net with fanout $k = 4$ and transition function $f(\lambda, y_1, y_2, y_3, y_4)$ can be used:

$$\begin{aligned} y_{r,k}^{(m-1)} &= f(\vec{0}, y_{r,k}^{(m)}, y_{r,k+2^{m-1}}^{(m)}, \vec{0}, \vec{0}), \\ y_{r+2^{n-m},k}^{(m-1)} &= f(\epsilon_m^k, \vec{0}, \vec{0}, y_{r,k}^{(m)}, y_{r,k+2^{m-1}}^{(m)}). \end{aligned}$$

Using a linearly activated hidden layer, this mode would be able to emulate the Fast Fourier Transformation. In general it would be possible to train a net to produce outputs that base on (parts of) the Fourier coefficients – without the need of computing them all through preprocessing the whole input. ϵ_m^k being a complex number is not a problem because gradient descent can also be used to train recursive nets using complex weights. The error function must have constant imaginary part 0 so it cannot be assumed holomorphic. Therefore no complex differentiation is used. Instead the outputs real and imaginary part is partially differentiated into real and imaginary part of its inputs, creating a set of 4 equations. By enforcing the Cauchy-Riemann conditions, a compact expression for the gradient can be acquired. Details can be found in [GM07], for example.

Using a net that has $\vec{0}$ as initial context and stationary point this mode could also be applied to sequences of length $\neq 2^n$ by simply using the initial context at missing positions.

Enhancing and modifying existing methods

In general, the periodic mode can be easily modified to not use implied $\vec{0}$ for inner labels but, for example, cylindrical coordinates by assuming the (sub)trees to exist on a cylindric surface. This would be suitable for a periodic target output that should not be translation invariant. For a periodic target output that is supposed to be translation invariant, the Euclidean coordinates of points distributed on the surface of a cone could be used. For this let all trees of the same depth have the same distance to the tip of the cone and the trees at the top-most level closest to the tip (thereby having almost the same coordinates).

Using non-zero vectors can also be done for the recursive or D&C mode. In all cases, the input label dimension should be increased for containing these coordinates.

All modes can be generalised to deal with d-dimensional inputs. For D&C restructuring, the input from $\mathbb{R}^{N_1 \times \ldots \times N_m}$ could be considered a sequence of length N_1, using the $x_{i,[N_2/2],\ldots,[N_m/2]}$ as labels and using child positions 1 and 2 within the output tree. When the length N_1 is exhausted, within each subtree belonging to i at the positions 3 and 4 the labels from the sequence $x_{i,j,[N_3/2],\ldots,[N_m/2]}$ with fixed i and varying j are processed, etc. Note that the pointwise length/width of the subsequence/hyperplane is allowed to differ due to the D&C mode adopting to each part individually.

The recursive mode bases on the idea of processing input windows of length k^d (here mostly $k = 2$). This can be generalised to two-dimensional data with another parameter l and using a recursive net with fanout $k \cdot l$. Now the recursion can be done over input windows of size $k^d \times l^d$. This directly transfers to the periodic restructuring: if all parameters are assumed to be 2, then for n-dimensional input hypercubes of size 2^{dn} are the input windows underlying the processing of a recursive net of fanout 2^n.

The Bidirectional D&C mode on trees can be defined as to create trees with labelled edges where the first edge receives label $(1, 0)$, the second $(0, 1)$ and the third edge, connected to the root-path, receives a label $(0.5, 0.5)$ for example. This way, the amount of free parameters is not increased through bidirectional restructuring.

10.5 Pattern sets

The focus on the pattern sets used in this thesis was on classification tasks. The periodic mode offers a computationally inexpensive way to deal with, for example, speech recognition or protein secondary structure prediction. It should be of

high interest to examine this restructuring mode on those problems. For speech recognition for example, error injection (see above) in all layers of the graph makes sense by specifying fraction of a certain phoneme that is contained in the underlying context.

The protein secondary structure prediction would be an application of the periodic mode for two-dimensional data. However, it could also be simply applied one-dimensional for predicting the protein folding process, that is, how the actual protein changes its shape. This is due to the periodic mode enabling a non-causal sequence-to-sequence prediction. If it were a "causal transducer", positions of one atom could only depend on the positions of its left neighbour, which obviously cannot properly reflect the physical background. This can be done by learning to map the sequence of atoms and their positions of the protein to a sequence of vectors that indicate how the atom would move (more precisely: the vectors indicate the force that the surrounding applies to the atom). When the input is a protein that has its stable form, the output would be a sequence of zeros (indicating that it is stable). This fact could be used to speed up the training process by starting the training with those stable proteins and applying special formulas.

Sparse data and (in)significant labels

Due to the nature of a recursive net, positions in an input tree, that do not contain a subtree, are numerically replaced with an initial context. While for sequential data this can only happen at the beginning of the sequence, for trees this can happen arbitrarily often. Therefore, by restructuring a sequence into a tree, a natural way of dealing with missing labels is possible.

Assume a sequence is given where some of its labels are uncertain due to reading errors or data loss in the physical system from where the data is taken. When processing this data as a sequence, missing labels must be replaced with a default value in order to keep the shape (length) of the sequence. The recursively or periodically restructured sequence however can just use $\perp$, that is, delete the leaf that would contain the missing label.

This fact could – in reverse – be used to deliberately remove elements from a sequence after successfully training on it. When the classifying net still produces correct results, the deleted element might have been insignificant.

10.6 Architectures

The restructuring methods do not impose any specific recursive net architecture that must be used. Though to some degree they might be a replacement for contextual architectures or bidirectional nets, they can still be combined with them. This has been shown through *bidirectional* Divide and Conquer, which, additionally, could still be used on a contextual cascade correlation architecture (see below).

Cascade Correlation and RTRL

Using RTRL for gradient descent is computationally inefficient in comparison to BPTT. However, for training the periodic mode, RTRL is the only way to avoid quadratic costs regarding the input length. For this reason, using Cascade Correlation as an architecture would be an approach to reduce the computational costs. Even though all weights are trained once, for each step during the creation of the net, only the recently added weights are trained while all others are frozen. Since for RTRL only the free parameters (the new weights) are a part of the gradient, the computational costs would considerably decrease.

Exponentially growing context through Contextual Cascade Correlation

Since data restructuring does not impose any limits to the network architecture, as long as it is defined as a recursive net, also Contextual Cascade Correlation ([MSS04]) could be used. This would have fascinating effects: when the $n+1$-th neuron is added into the cascade architecture, this neuron receives input from the n-th neuron, but not only by its activations from childtrees (q^{-1}), but also from parent trees (via q^{+1}). Compare figure 4.4 for example (not containing connections regarding q^{+1}). In the periodic mode (compare figure 6.6), the activations at level L are functions from a subsequence of length 2^L of the input sequence. This means: when activating a leaf (at level 0 of the periodic mode), the recently added $n+1$-th neuron has a visible context of length 2^n because it indirectly is a function of the first neurons activation at level n. The context size for activations in a leaf is growing exponentially with the size of the net.

Spiral Recurrent Networks

Spiral Recurrent Networks ([GSK07]) would be very interesting to examine with the periodic mode. The weight matrices for Spiral Recurrent Networks have a restricted eigenvalue spectrum. If they are generalised to recursive nets and the weight tensors can be constructed as to fulfil the invertibility prerequisites described in chapter 6.6.2.2 they could be well suited for a protein folding task. The first layer of the weight tensor could be defined to be close to the unit matrix in order to create the

identity function for protein structures that reached a stable point, while the second layer must learn to have influence mainly when the structure is supposed to change its shape which presumably depends on the neighbourhood/context which - in this application - would be accessed over the right child (χ_2) of a subtree.

Bibliography

[Arn08] Sven Arnhold. “Long Short-Term Memory und rekursive Netze”. Diplomarbeit. Institut für Informatik, TU Clausthal, Aug. 2008.

[Bal+01] Pierre Baldi et al. “Bidirectional Dynamics for Protein Secondary Structure Prediction”. In: *Sequence Learning*. Springer, 2001, pp. 80–104.

[Bia+05] Monica Bianchini et al. “Recursive neural networks for processing graphs with labelled edges: theory and applications”. In: *Neural Networks* 18.8 (2005), pp. 1040–1050.

[BP03] Pierre Baldi and Gianluca Pollastri. “The Principled Design of Large-Scale Recursive Neural Network Architectures-DAG-RNNs and the Protein Structure Prediction Problem”. In: *Journal of Machine Learning Research* 4 (2003), pp. 575–602.

[BSF94] Yoshua Bengio, Patrice Simard and Paolo Frasconi. “Learning Long-Term Dependencies with Gradient Descent is Difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1.1.41.7128.

[CF01] Rafael C. Carrasco and Mikel L. Forcada. “Simple Strategies to Encode Tree Automata in Sigmoid Recursive Neural Networks”. In: *IEEE Transactions on Knowledge and Data Engineering* 13 (2001), pp. 148–156.

[GFS07] Alex Graves, Santiago Fernandez and Jürgen Schmidhuber. “Multi-Dimensional Recurrent Neural Networks”. In: *Proceedings of the 2007 International Conference on Artificial Neural Networks*. 2007.

[GK96] Christoph Goller and Andreas Küchler. “Learning Task-Dependent Distributed Representations by Backpropagation Through Structure”. In: *IEEE International Conference on Neural Networks* 1 (1996), pp. 347–352. DOI: 10.1109/ICNN.1996.548916.

[GM07] Su Lee Goh and Danilo P. Mandic. “An augmented CRTRL for complex-valued recurrent neural networks”. In: *Neural Networks* 20.10 (2007), pp. 1061–1066.

[GS05] Alex Graves and Jürgen Schmidhuber. “Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures”. In: *Neural Networks* 18.5-6 (2005), pp. 602–610. URL: ftp://ftp.idsia.ch/pub/juergen/nn_2005.pdf.

[GSC00] F. A. Gers, J. Schmidhuber and F. Cummins. “Learning to Forget: Continual Prediction with LSTM”. In: *Neural Computation* 12.10 (2000), pp. 2451–2471.

[GSK07] Huaien Gao, Rudolf Sollacher and Hans-Peter Kriegel. “Spiral Recurrent Neural Network for Online Learning”. In: *ESANN 2007, 15th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 25-27, 2007, Proceedings*. 2007, pp. 483–488. URL: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2007-39.pdf.

[GSS02] F. A. Gers, N. N. Schraudolph and J. Schmidhuber. “Learning Precise Timing with LSTM Recurrent Networks”. In: *Journal of Machine Learning Research* 3 (2002), pp. 115–143.

[Ham07] Barbara Hammer. “Neural Computation”. Lecture Notes. TU Clausthal, Germany. 2007.

[Ham97] Barbara Hammer. *On the Generalization Ability of Simple Recurrent Neural Networks*. Tech. rep. Universität Osnabrück, 1997.

[Ham99] Barbara Hammer. “Learning with Recurrent Neural Networks”. PhD thesis. Universität Osnabrück, 1999.

[HB05] John Hawkins and Mikael Boden. “The Applicability of Recurrent Neural Networks for Biological Sequence Analysis”. In: *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 2.3 (2005), pp. 243–253. ISSN: 1545-5963. DOI: 10.1109/TCBB.2005.44. URL: http://dx.doi.org/10.1109/TCBB.2005.44.

[HHO07] Sepp Hochreiter, Martin Heusel and Klaus Obermayer. “Fast model-based protein homology detection without alignment”. In: *Bioinformatics* 23.14 (2007), pp. 1728–1736.

[HMS07] Barbara Hammer, Alessio Micheli and Alessandro Sperduti. “Perspectives of Neural-Symbolic Integration”. In: ed. by Hammer and Hitzler. Springer, 2007. Chap. Adaptive Contextual Processing of Structured Data by Recursive Neural Networks: A Survey of Computational Properties, pp. 67–94.

[Hor89] K. Hornik. “Multilayer Feedforward Networks are Universal Approximators”. In: *Neural Networks* 2 (1989), pp. 359–366.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[IH00] Christian Igel and Michael Hüsken. “Improving the Rprop Learning Algorithm”. In: *Proceedings of the Second International Symposium on Neural Computation*. ICSC Academic Press, 2000, pp. 115–121.

[Jac05] Henrik Jacobsson. “Rule Extraction from Recurrent Neural Networks: A Taxonomy and Review”. In: *Neural Computation* 17.6 (2005), pp. 1223–1263.

[JS98] Merten Joost and Wolfram Schiffmann. “Speeding Up Backpropagation Algorithms by Using Cross-Entropy Combined with Pattern Normalization”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.2 (1998), pp. 117–126.

[KP90] John F. Kolen and Jordan B. Pollack. “Back Propagation is Sensitive to Initial Conditions”. In: *NIPS*. 1990, pp. 860–867.

[Log00] Beth Logan. “Mel Frequency Cepstral Coefficients for Music Modeling”. In: *In International Symposium on Music Information Retrieval*. 2000.

[May+06] Hermann Georg Mayer et al. “A System for Robotic Heart Surgery that Learns to Tie Knots Using Recurrent Neural Networks”. In: *IROS*. 2006, pp. 543–548.

[MSS04] A. Micheli, D. Sona and A. Sperduti. “Contextual Processing of Structured Data by Recursive Cascade Correlation”. In: *IEEE Transactions on Neural Networks* 15.6 (2004), pp. 1396–1410. ISSN: 1045-9227. DOI: 10.1109/TNN.2004.837783.

[RB92] Martin Riedmiller and Heinrich Braun. “RPROP - A Fast Adaptive Learning Algorithm”. In: *Proceedings of the International Symposium on Computer and Information Science VII*. 1992, pp. 279–285.

[SB07] Stoer and Bulirsch. “Numerische Mathematik I”. In: ed. by Freund and Hoppe. Springer, 2007. Chap. 2.3, pp. 81–82.

[Sca+09] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80.

[Sch92] Jürgen Schmidhuber. “A Fixed Size Storage $O(n^3)$ Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks”. In: *Neural Computation* 4.2 (1992), pp. 243–248.

[Sch99] Nicol N. Schraudolph. “A Fast, Compact Approximation of the Exponential Function”. In: *Neural Computation* 11.4 (1999), pp. 853–862. DOI: 10.1162/089976699300016467).

[SG98] T. Schmitt and C. Goller. “Relating Chemical Structure to Activity: An Application of the Neural Folding Architecture”. In: *Engineering Applications of Neural Networks* (1998).

[Soc+11] Richard Socher et al. “Parsing Natural Scenes and Natural Language with Recursive Neural Networks”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. Ed. by Lise Getoor and Tobias Scheffer. ICML ’11. Bellevue, Washington, USA: ACM, 2011, pp. 129–136. ISBN: 978-1-4503-0619-5.

[Son92] Eduardo D. Sontag. “Feedforward Nets for Interpolation and Classification”. In: *Journal of Computer and System Sciences* 45 (1992), pp. 20–48.

[SP97] Mike Schuster and Kuldip K. Paliwal. “Bidirectional Recurrent Neural Networks”. In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[SS97] Alessandro Sperduti and Antonina Starita. “Supervised Neural Networks for the Classification of Structures”. In: *IEEE Transactions on Neural Networks* 8 (1997), pp. 714–735.

[Sta03] Burkhard Stackelberg. “Konstruktionsverfahren vorwärtsgerichteter neuronaler Netze”. ger. PhD thesis. Holzgartenstr. 16, 70174 Stuttgart: Universität Stuttgart, 2003. URL: `http://elib.uni-stuttgart.de/opus/volltexte/2003/1444`.

[Wer90] P. J. Werbos. “Backpropagation Through Time: What It Does and How to Do It”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: `10.1109/5.58337`.

[WZ89] Ronald J. Williams and David Zipser. “A Learning Algorithm for Continually Running Fully Recurrent Neural Networks”. In: *Neural Computation* 2 (1989), pp. 270–280.

[Zel94] Andreas Zell. “Simulation Neuronaler Netze”. In: Addison Wesley, 1994. Chap. 9, pp. 124–126.